НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«На правах рукопису»
УДК 004.415.2

До захисту допущено:
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Програмне забезпечення інформаційно-комунікаційних систем»
зі спеціальності 121 «Інженерія програмного забезпечення»
на тему: «Автоматизована система моніторингу та керування обчислювальними ресурсами у Kubernetes кластері»

Виконав:
студент VI курсу, групи ІТ-93мп
Омельченко Віталій Вікторович _____

Керівник:
завідувач каф. АУТС, професор, д.т.н.
Ролік Олександр Іванович _____

Рецензент:
доцент каф. ТК, к.т.н.
Ткач Михайло Мартинович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Програмне забезпечення інформаційно-комунікаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____20__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Омельченку Віталію Вікторовичу

1. Тема дисертації «Автоматизована система моніторингу та керування обчислювальними ресурсами у кластері Kubernetes», науковий керівник дисертації Ролік Олександр Іванович, д.т.н., професор, завідувач кафедри АУТС, затверджені наказом по університету від «26» 10 2020 р. №3132-с

2. Термін подання студентом дисертації _____

3. Об'єкт дослідження: процес розподілення обчислювальних ресурсів у кластері Kubernetes.

4. Вихідні дані: технічна література про Kubernetes, розробку додатків до нього, методи масштабування, документація Kubernetes, документація операційної системи Linux, матеріали мережі Інтернет, що стосуються теми роботи.

5. Перелік завдань, які потрібно розробити: опис предметної області, аналіз існуючих рішень, аналіз та вибір технічних засобів для реалізації системи, розроблення програмного рішення, тестування роботи системи, графічний матеріал.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: діаграма розгортання, діаграма компонентів, діаграма прецедентів, діаграма сутностей,

блок-схема алгоритму роботи системи, блок-схема алгоритму роботи планувальника, схему отримання метрик, схема тестового середовища, структурна схема.

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Огляд предметної області	07.09.2020 р.	
2.	Аналіз існуючих рішень	21.09.2020 р.	
3.	Аналіз та вибір технічних засобів для реалізації системи	28.09.2020 р.	
4.	Розгортання середовища для розробки та тестування	01.10.2020 р.	
5.	Розроблення програмного рішення	30.11.2020 р.	
6.	Тестування розробленої системи	12.11.2020 р.	
7.	Розроблення стартап – проекту	19.11.2020 р.	
8.	Оформлення текстової документації	01.11.2020 р.	
9.	Подання готової роботи	03.12.2020 р.	

Студент Віталій ОМЕЛЬЧЕНКО

Науковий керівник Олександр РОЛІК

РЕФЕРАТ

Магістерська дисертація на тему «Автоматизована система моніторингу та керування обчислювальними ресурсами у кластері Kubernetes».

Робота містить 112 с. тексту, 61 рисунок, 14 таблиць, 33 джерела та 10 додатків.

Сьогодні існує масова тенденція переходу від монолітної до мікросервісної архітектури, зокрема, на основі оркестратора Kubernetes. Керування кластером Kubernetes стає значно складнішим з ростом кількості вузлів та мікросервісів, тому процес управління кластером потребує автоматизації, зокрема процес керування обчислювальними ресурсами.

Об’єктом розробки є автоматизована система моніторингу та керування обчислювальними ресурсами у кластері Kubernetes.

Метою магістерської дисертації є оптимізація використання обчислювальних ресурсів у кластері, а саме мінімізація збиткового резервування ресурсів, що в свою чергу призведе до зменшення фінансових витрат на обслуговування та розширення кластеру.

Предметом дослідження є алгоритми розподілення обчислювальних ресурсів у кластері Kubernetes.

В даній роботі проектується підсистема для моніторингу та керування обчислювальними ресурсами у кластері Kubernetes з використанням мови програмування Go та фреймворку Operator SDK, а також алгоритми для обрахунку оптимальних квот для ресурсів. На основі результатів розробки створено план для стартап-проекту.

Ключові слова: Kubernetes, мікросервіси, мікросервісна архітектура, управління ресурсами.

ABSTRACT

Master's dissertation on "Automated system for monitoring and managing computing resources in the Kubernetes cluster."

The work contains 112 p. of text, 61 figures, 14 tables, 33 references and 10 appendices.

Today there is a massive trend of migration from monolithic to microservice architecture, in particular, on the basis of the Kubernetes orchestrator. Kubernetes cluster management becomes much more complex as the number of nodes and microservices grows, so the cluster management process requires automation, including the process of managing computing resources.

The object of research is an automated system for monitoring and managing computing resources in the Kubernetes cluster.

The purpose of the master's dissertation is to automate the process of managing computing resources in the cluster, in particular, to provide optimal quotas for resources for microservices.

The subject of research are algorithms for computing resources in the Kubernetes cluster.

A subsystem for monitoring and managing computing resources in the Kubernetes cluster using the Go programming language and the Operator SDK framework has been developed, as well as algorithms for calculating optimal quotas for resources. A plan for a startup project was created based on the results of the development.

Keywords: Kubernetes, microservices, microservice architecture, resource management.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

API	Application user interface
CI/CD	Continuous integration та continuous delivery
CPU	Central processor unit
DevOps	Development operation
GCP	Google Cloud Platform
GKE	Google Cloud Engine
GRPC	Google remote procedure call
GUI	Graphical user interface
HTTP	HyperText transfer protocol
HTTPS	Secured HyperText transfer protocol
HyperThreading	Технологія Intel для виконання двох потоків на одному ядрі
OOM	Out of memory
QoS	Quality of service
REST	Representational state transfer
SSD	Solid state drive
ОС	Операційна система
Под	Pod в системі Kubernetes

ВСТУП

Стрімкий розвиток технологій та збільшення кількості як користувачів мережі Інтернет, так і різного роду розумних пристроїв постійно встановлюють нові та все вищі вимоги до обслуговуючих інформаційних систем. Веб-сервіси пропонують все більше можливостей та функцій, що стосуються різних сфер нашої діяльності. Саме через ці причини традиційна монолітна архітектура веб-додатків втрачає актуальність. Для того щоб підтримувати масштабованість та надійність постійно зростаючих інформаційних систем, розробники, як правило, переходять від монолітного стилю архітектури програмного забезпечення до мікросервісної архітектури.

Мікросервісна архітектура має багато переваг, серед яких гнучкість, ізольованість даних, масштабованість окремих сервісів, незалежність від мови програмування та платформи в контексті нових сервісів, відмовостійкість. Проте дана архітектура створює нові проблеми – залежність від мережі, розподілені транзакції, наявність значно більшої кількості компонентів, за якими необхідно слідкувати та підвищені вимоги до обчислювальних ресурсів у порівнянні з монолітною архітектурою. На фоні всіх інших проблем задача моніторингу та керування обчислювальними ресурсами може здаватися другорядною. Проте неправильне керування обчислювальними ресурсами кластера може призвести до значних економічних витрат, нестачі ресурсів у кластері та відмови сервісів. Саме тому проблема автоматизації моніторингу та управління обчислювальними ресурсами є дуже актуальною та потребує уваги.

Мікросервісна архітектура передбачає використання спеціальних систем – оркестраторів, які надають інструменти для автоматизації розгортання, масштабування та управління сервісами. Дані системи розміщують та балансують додатки на групі фізичних або віртуальних машин – кластері. Kubernetes – це один із найрозповсюдженіших оркестраторів, який включає всі необхідні інструменти для побудови мікросервісної архітектури незалежно від хмарної платформи. Саме в

контексті даної системи розглядається проблема моніторингу та керування обчислювальними ресурсами.

Найважливішими ресурсами кластера є оперативна пам’ять, процесорний час, ресурс мережі, пам’ять постійного зберігання (HDD, SSD). Постійна пам’ять є досить дешевою та легко піддається моніторингу, а мережевий ресурс рідко стає проблемою. Саме перші два є найбільш критичними, дорогими та найскладнішими у керуванні ресурсами.

Об’єктом дослідження даної магістерської дисертації є використання ресурсів пам’яті та процесора у кластері Kubernetes та алгоритм роботи оркестратора в умовах нестачі ресурсів.

Тема даної магістерської дисертації присвячена вирішенню проблеми автоматизації моніторингу використання обчислювальних ресурсів у кластері, а також їх керування.

Метою магістерської дисертації є оптимізація використання обчислювальних ресурсів у кластері, а саме мінімізація збиткового резервування ресурсів, що в свою чергу призведе до зменшення фінансових витрат на обслуговування та розширення кластеру. Дана мета досягається за рахунок автоматизації процесу моніторингу утилізації ресурсів та налаштування квот на використання пам’яті та процесорного часу.

Враховуючи стрімко зростаючу популярність мікросервісної архітектури з використанням Kubernetes та складність керування ресурсами в таких системах тема роботи є актуальною.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

У даному розділі розглядається мікросервісна архітектура, її принципи та особливості. Оскільки дана робота безпосередньо стосується роботи мікросервісних застосунків в рамках системи Kubernetes, то необхідно також ознайомитись з контейнеризацією, яка є невід'ємною частиною Kubernetes, а також принципами роботи самого Kubernetes. Додатково розглядається алгоритм роботи планувальника в даній системі, як важлива частина керування обчислювальними ресурсами у кластері.

1.1 Аналіз мікросервісної архітектури

В даному підрозділі розглядається мікросервісна архітектура, її переваги та недоліки та неявно порівнюється з монолітним підходом до створення систем.

1.1.1 Особливості мікросервісної архітектури

Мікросервісна архітектура – це підхід, при якому додаток будується модульним, де модулі – це набір невеликих сервісів, кожен з яких працює у власному ізольованому середовищі та, при потребі, спілкується з іншими використовуючи високорівневі протоколи міжсервісної взаємодії, наприклад, REST або GRPC.

Автор книги «Building Microservices: Designing Fine-Grained Systems» описує поняття "мікросервіс" всього одним реченням: "Кожен мікросервіс є автономним і повинен реалізовувати єдину бізнес-функцію" [2]. Отже мікросервіси побудовані навколо конкретних бізнес–потреб і розгортаються незалежно один від одного з використанням інтегрованого та спеціально автоматизованого середовища – CI/CD. Кожен мікросервіс може бути написаний на будь-якій мові, незалежно від інших, з використанням різних підходів та патернів.

На рисунку 1.1 розглядається абстрактний приклад мікросервісної архітектури.

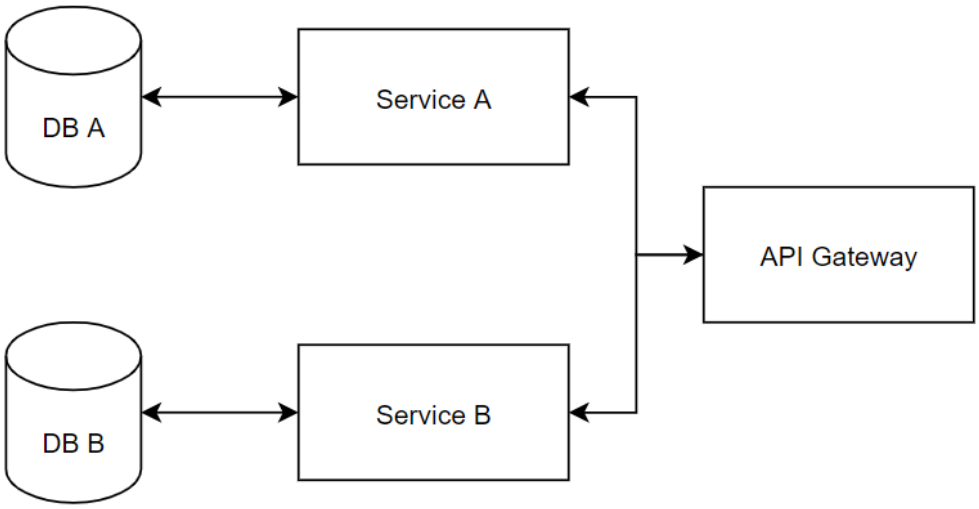

Рисунок 1.1 – Найпростіший приклад мікросервісної архітектури

На рисунку 1.1 зображено найпростіший приклад мікросервісної архітектури. На діаграмі бачимо 5 складових:

- API Gateway – це також мікросервіс, але він не виконує жодної бізнес задачі. Його основна функція – це надати загальний API для користувача, агрегуючі інтерфейси сервісів А та В, аналогічно до патерну фасад в програмуванні [3];
- сервіси А та В мікросервіси, що виконують власні бізнес функції, та надають API для звертань через API Gateway [4];
- бази даних – кожен мікросервіс зберігає дані в ізольованій базі даних. Якщо йому необхідні дані з іншого мікросервісу, то отримує їх через API. Використання прямих підключень до баз сторонніх сервісів є поганим тоном та може призвести до проблем при міграції схеми бази даних;

Цю схему легко застосувати у будь-якій сфері. Нехай API Gateway – це API Telegram, а мікросервіс А – це додаток для обміну повідомлення, мікросервіс Б – для роботи з контактами користувача. На рисунку 1.2 розглядається конкретна мікросервісна архітектура, а саме архітектура сервісу таксі.

На рисунку 1.2 можемо побачити тісну взаємодію групи мікросервісів, де кожен мікросервіс виконує лише одну функцію в системи. Наприклад, мікросервіс billing відповідає за обрахунок балансів та проведення транзакцій, а мікросервіс notifications відповідає за надсилання повідомлень.

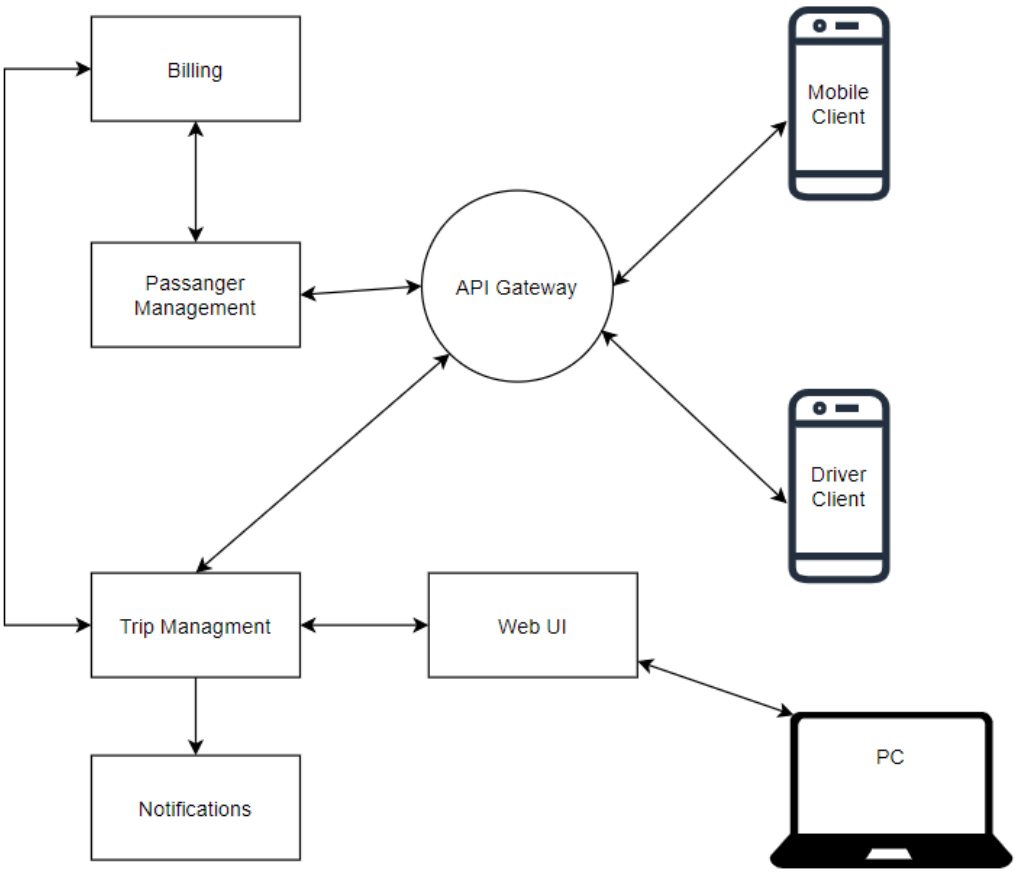

Рисунок 1.2 – Архітектура сервісу таксі

На рисунку 1.2 бачимо значно складнішу архітектуру, де маємо кілька точок входу – API Gateway та Web UI, сервіси виконують лише одну функцію та взаємодіють між собою. Кожна функціональна область програми тепер реалізована власним мікросервісом.

Далі розглядається чому існує необхідність розробляти саме програми, які покривають лише одну функцію у всій системі. Кодова база постійно зростає в процесі додавання нових функцій у нашому додатку. Через деякий час функцій може бути настільки багато, що буде складно знайти всі місця, де потрібно внести зміни під час, наприклад, рефакторингу проекту. Незважаючи на використання чітких підходів під час розробки монолітних додатків, використання загальноприйнятих патернів та врегульованих принципів на проекті, іноді доводиться пожертвувати ними задля додавання нового функціоналу. Кількість коду, пов’язаного з подібними ситуаціями, починає зростати, ускладнюючи виправлення помилок або робить додавання нового функціоналу значно складнішим [2]. Під час розробки монолітної архітектури

розробники часто намагаються виділити домени розробки, створюючи абстракції або модулі. Домени – прагнення об’єднати зв’язаний код – є невід’ємною частиною мікросервісної архітектури [5]. Це підкріплюється визначенням Роберта К. Мартіна «принципа єдиної відповідальності», в якому сказано: «Об’єднайте ті речі, які змінюються з тієї самої причини, і розділіть ті речі, які змінюються з різних причин» [6].

Мікросервіси використовують такий самий підхід під час розділення функціоналу на мікросервіси. Розробники об’єднують зв’язаний код в окремий мікросервіс, визначаючи межі домену. Таким чином, одна функція моноліту замінюється одним мікросервісом. Між програмним викликом та викликом API мікросервісу не така вже і велика різниця.

Переваги даної архітектури:

- гнучкість - оскільки мікросервіси розгортаються окремо, то додавати новий функціонал в систему або виправляти помилки значно простіше та не потребує перерозгортання інших мікросервісів. Також можливо швидко скасувати оновлення у разі проблем після розгортання. У монолітному додатку, наприклад, помилку у одному модулі призвела би до скасування оновлення для всіх функцій релізу;
- спеціалізовані та невеликі команди розробників. Команда відповідає лише за свій мікросервіс і має базу знать по ньому. Великі команди є менш продуктивними, оскільки це призводить до інтенсивної комунікації та ускладненого менеджменту;
- кодова база. Незважаючи на використання чітких підходів під час розробки монолітних додатків, використання загальноприйнятих патернів та врегульованих принципів на проекті, іноді доводиться пожертвувати ними задля додавання нового функціоналу. В мікросервісній архітектурі така проблема відсутня, оскільки кодова база повністю ізольована від інших;
- технології – розробники можуть обирати ті інструменти, які найкраще підходять для поставленої задачі. Це стосується мови програмування, фреймворків, бібліотек та баз даних, що зображено на рисунку 1.3;

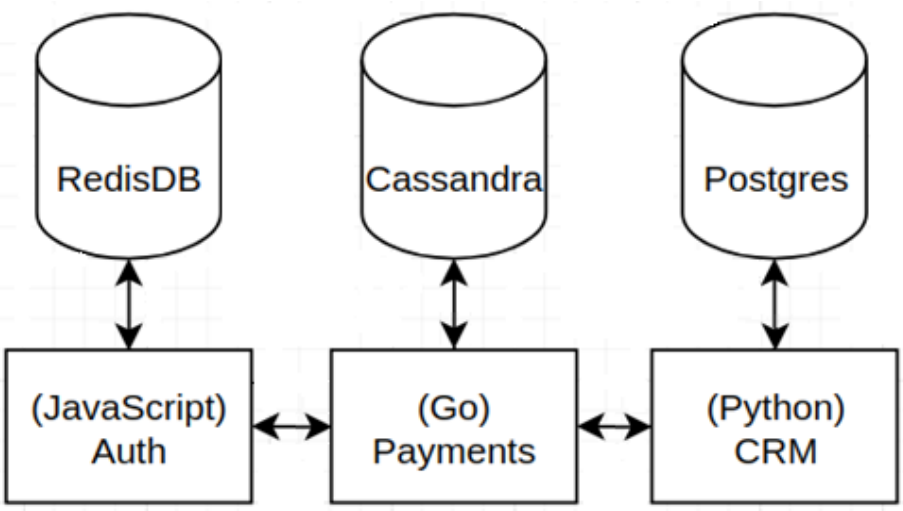

Рисунок 1.3 – Гнучкість технологій

- ізоляція відмов - якщо в одному з мікросервісів сталася помилка та він не працює, то мікросервіси не зв’язані з ним будуть працювати в нормальному режимі. Так, наприклад, якщо в базі монолітного додатка сталася відмова, то працювати не буде весь функціонал;
- масштабованість - є можливість збільшити кількість одиниць лише конкретного мікросервісу під час навантаження на нього. Або навпаки, зменшити до мінімуму кількість одиниць, якщо навантаження відсутнє;

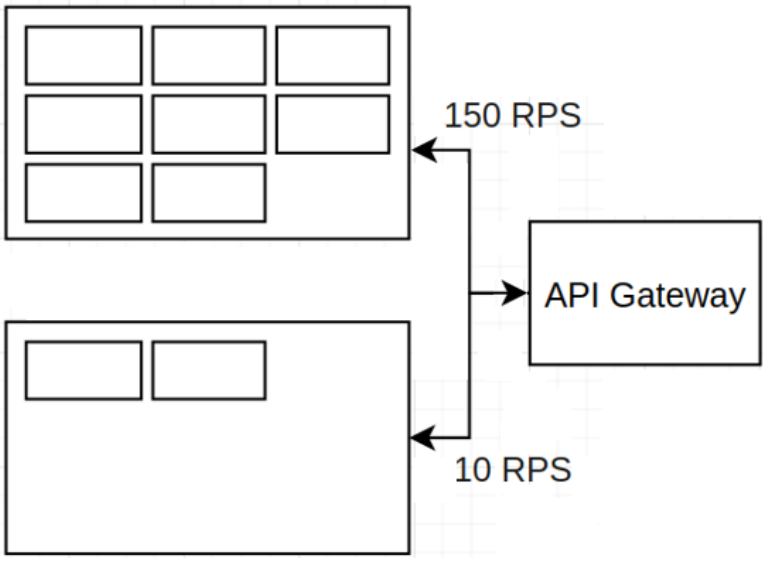

Рисунок 1.4 – Приклад масштабування

- ізоляція даних – кожен мікросервіс має свою базу даних. Таким чином міграція схеми бази даних одного мікросервісу ніяк не може призвести до відмов інших мікросервісів.

Всі переваги, перераховані вище, не даються безкоштовно. Є декілька актуальних проблем при використанні мікросервісної архітектури.

По-перше, значно зростає складність системи. Додаток має значно більше активних частин, ніж аналогічний монолітний додаток. Хоча кожен мікросервіс простіший, але вся система в цілому є складнішою.

По-друге, це помилки та затримки мережі. Комунікація між мікросервісами здійснюється через мережу, а передача по мережі не є повністю надійною. Крім того, якщо ланцюг вигликів мікросервісів може бути занадто довгим, тому затримка може бути проблемою. Необхідно ретельно проектувати API. Також необхідно по максимуму застосовувати асинхронну модель зв’язку [7].

Наступне, децентралізація. Децентралізований підхід до побудови мікросервісів має переваги, але це також може створити проблем. Наприклад, у розробників може бути стільки різних мов та фреймворків, що додаток стає занадто складно підтримувати. Тому необхідно встановити деякі загальні стандарти проектування. Крім того, для використання мікросервісів потрібна зріла культура DevOps.

1.1.2 Засоби реалізації мікросервісної архітектури

У даному розділі розглядаються необхідні технічні засоби для реалізації мікросервісної архітектури – контейнеризація, оркестрація та CI/CD, а також обґрунтовується чому необхідна контейнеризація або віртуалізація.

По-перше, це ізольоване середовище виконання. Віртуалізація надає розробникам можливість створювати власні передбачувані середовища виконання, які ізольовані від інших програм та абстрагуватись від особливостей фізичних машин. Контейнери включають лише ті програмні залежності, які необхідні додатку, такі як конкретні версії версія компілятора для мови програмування та версія деякої

системної бібліотеки. Віртуалізація гарантує що середовище для роботи додатку буде налаштовано саме як того потребує розробник, незалежно від того, де додаток розгортається. Все описане вище призводить до підвищення ефективності розробників та адміністраторів, адже вони витрачають значно менше часу та сил на відлагодження та діагностування проблем, спричинених відмінностями в середовищах запуску, а більше часу починають використовувати для запуску нового функціоналу для клієнтів. Це також означає менше несподіваних помилок, оскільки розробники можуть бути впевненими, що тестове середовище відповідає реальному.

По-друге, це незалежність від платформи та операційної системи. Віртуальні машини можуть працювати практично в будь-якій операційній системі, значно полегшуючи розробку та розгортання – це може бути Windows, Linux або MacOS, або це можуть бути хмарні платформи – Google Cloud Platform, AWS або DigitalOcean та незалежно від архітектури – x86 чи ARM.

По-третє, це ізоляція. Контейнери мають власний віртуальний центральний процесор, пам'ять, сховище та мережеві ресурси на рівні операційної системи, ізолюючі додаток від інших програм [8].

Мікросервісна архітектура не вимагає використання контейнеризації. Наприклад, компанія Netflix довгий час ігнорувала тенденцію використання контейнерів та використовувала екземпляри AWS напряму. Але більшість організацій, які мігруються на мікросервісну архітектуру, знайдуть контейнери більш зручним способом запуску своїх додатків [10].

Розміщення додатку у одному екземплярі операційної системи, допоможуть досягти кращих показників утилізації серверних ресурсів. Саме тому якщо організація використовує контейнеризацію мікросервісних додатків у хмарних платформах, таких підхід допоможе в зменшенні витрат. Якщо ж говорити про власний кластер, то в такому випадку запас обчислювальних ресурсів виросте порівняно з використанням віртуалізації.

Більш детально переваги та недоліки контейнеризації та віртуалізації розглядаються у наступному розділі.

Додаток може включати десятки або сотні мікросервісів, а ті в свою чергу можуть мати десятки контейнерів. Тому для керування необхідні спеціалізовані рішення – оркестратори.

Основною задачею оркестратора є управління контейнерами. Оркестратор запускає, конфігурує, розміщує між вузлами користувацькі контейнери. Крім того налаштовує балансування навантаження між контейнерами. Також, оркестратор постійно слідкує за всіма запущеними контейнерами та перезапускає контейнери, що не відповідають або займають занадто багато ресурсів [11].

Один із важливих компонентів мікросервісної архітектури – це CI/CD. Коли говориться про CI / CD, мається на увазі декілька процесів – безперервна інтеграція, безперервна доставка.

Постійна інтеграція забезпечує, що кожна нова функція, яка потрапила до головної гілки пройшла все необхідні тести та перевірки. Таким чином автоматизується перевірка якості коду та його відповідності всім вимогам на проекті. Безперервна доставка забезпечує, що будь-яка нова функція, яка пройшла процес CI, автоматично може розгорнутися у середовищі для відлагодження та перевірки. Розгортання в головному середовищі може вимагати деяких ручних перевірок та дій, але також частково автоматизується. Мета полягає в тому, що ваш код завжди є готовим до розгортання у основному середовищі [27].

Аналіз переваг використання CI / CD в мікросервісній архітектурі:

- кожна команда може самостійно та незалежно розробляти новий функціонал в мікросервісах, якими вони володіють, не зачіпаючи та не порушуючи роботу інших команд;
- перш ніж нова версія мікросервісу розгорнеться на основному середовищі, вона буде доступною для перевірки в середовищі розробки та тестування;
- поряд із попередньою версією можливо розгорнути попередню версія для часткової міграції або A/B тестування.

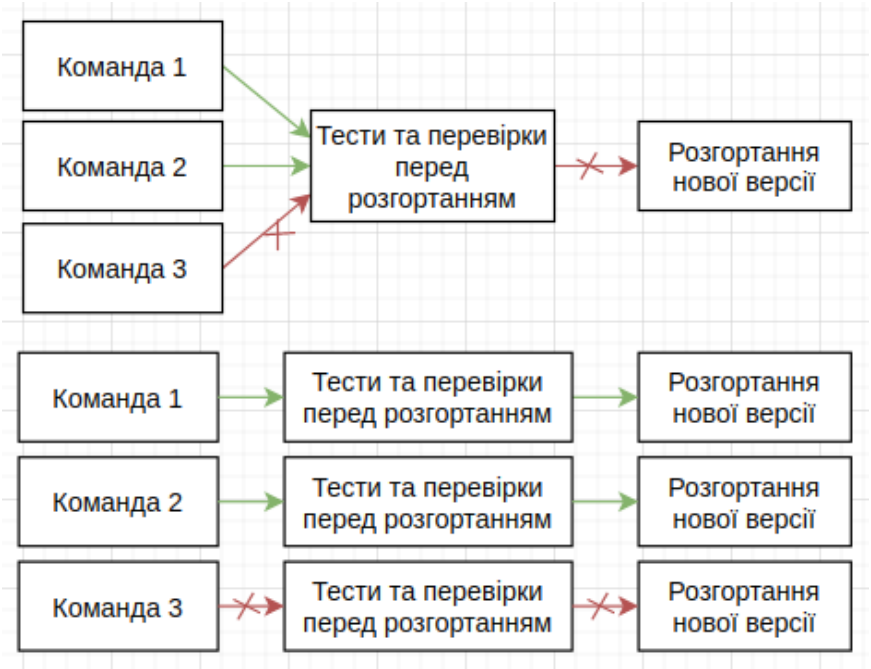

Рисунок 1.5 – Порівняння CI\CD в моноліті та мікросервісах

1.2 Підходи до контейнеризації застосунків

Буває, що компанія тримає систему з декількох комп'ютерів, яка їй фактично не потрібна. Типовим прикладом може послужити наявність в компанії поштового сервера, веб-сервера і FTP-сервера, серверів для зберігання даних та інших. І всі вони працюють на різних комп'ютерах в єдиній стійці обладнання, з'єднані мережею. Однією з причин запуску всіх цих серверів на окремих фізичних машинах може послужити те, що одна машина не може впоратися з покладеним на неї навантаженням, а інша потребує особливою надійністю: керівництво компанії просто не вірить в те, що операційна система може працювати без збоїв 24 години на добу 365 днів на рік. А якщо кожна служба поміщена на окремий комп'ютер, то збій одного з серверів принаймні не вплине на роботу інших серверів. Також при цьому легше вирішуються питання безпеки. Навіть якщо зловмисник проникне на веб-сервер, то одночасно з цим він не отримає доступ до сервісу пошти. Хоча завдяки цьому досягаються ізоляція додатків і стійкість до збоїв, таке рішення є досить дорогим і важко керованим, оскільки в ньому задіяно безліч фізичних машин. Проте існує технологія віртуалізації, яка здатна забезпечити більшість вимог описаних вище [12].

Віртуалізація – це процес запуску віртуального екземпляра операційної, системи на рівні, абстрагованому від апаратного забезпечення. Найчастіше це стосується запуску декількох операційних систем на комп’ютерній системі одночасно. Для програм, що працюють поверх віртуалізованої машини, емулюється, що вони перебувають на власній виділеній фізичній машині, де операційна система, бібліотеки та інші програми є унікальними для гостьової віртуалізованої системи та ніяким чином не залежать від головної операційної системи, яка працює під гостьовою [13].

Віртуалізація дозволяє більш ефективно використовувати ресурси, оскільки дозволяє розгорнути декілька незалежних між собою програм на одній фізичній машині, які працюють паралельно. Більш того, віртуалізація дозволяє призупиняти та зберігати стан віртуальної машини для того, щоб мігрувати її на інший фізичний сервер, якщо, наприклад, на поточній машині недостатньо ресурсів. Крім того, апаратне забезпечення є дорогим, а цей підхід дозволяє ефективно використовувати весь апаратний ресурс.

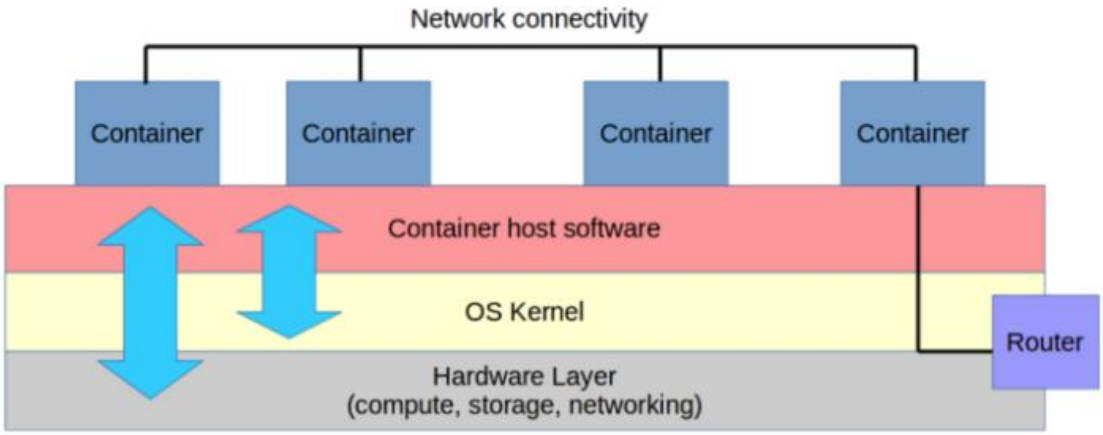

Рисунок 1.6 – Схема роботи контейнерів [13]

Проте в мікросервісній архітектурі більш поширеним підходом є використання контейнерів. Контейнер – це ізольована, легка операційна система для запуску додатку в операційній системі на фізичній або віртуальній машині. Контейнери спираються на ядро операційної системи основної машини і містять мінімальну

кількість додатків, та деякі API-інтерфейси. На рисунку 1.6 зображено схему роботи Docker-контейнерів [13].

На даній схемі можна побачити, що контейнеризація не емулює апаратну частину, а використовує апаратні ресурси основної ОС. А всі виклики контейнеризованої операційної системи транслюються в виклики до основної ОС.

Таблиця 1.1 – Порівняння віртуалізації та контейнеризації

	Віртуалізація	Контейнеризація
Ізоляція	Повна	Часткова – лише на рівні процесів та файлової системи
Швидкість запуску	Повільно, оскільки ініціалізується повноцінна ОС	Швидко
Продуктивність додатків	Зменшена через складність апаратного емулювання	Так, якби программа працювала на основній ОС
Розмір	Розмір як у повноцінної ОС	Мінімальний

Хоча віртуальні машини спрощують ізоляцію середовищ виконання, використання окремих віртуальних машин для кожного мікросервісу вимагає великих ресурсних витрат, оскільки кожна віртуальна машина вимагає власної операційної системи, що описано в таблиці 1.1. Хоча технічно можливо запустити декілька програм у одній віртуальній машині, це створює ризик того, що компоненти можуть конфліктувати між собою. Завантаження декількох мікросервісів в одну віртуальну машину спричиняє ту саму проблему, яка може виникнути під час запуску декількох програм на одному фізичному сервері.

Використання віртуальних машин також накладає обмеження на продуктивність. Кожна віртуальна машина, яка запускає власне середовище

виконання та операційну систему, використовує ресурси машини, які б могли бути використані для роботи додатків.

Навпаки, контейнери виконують ізоляцію виконання на рівні операційної системи. Тут один екземпляр операційної системи може підтримувати кілька контейнерів, кожен з яких працює у своєму окремому середовищі виконання. Запускаючи декілька компонентів в одній операційній системі, зменшуються накладні витрати, звільняючи ресурси для роботи мікросервісів. Тому з точки зору ефективності, контейнери є набагато кращим вибором для мікросервісної архітектури, ніж віртуальні машини.

В контейнерах є можливість використовувати контрольні групи (cgroups), щоб ізолювати середовище виконання для додатку, гарантуючи, що кожен додаток не може впливати на роботу інших [14]. Ця можливість, що надається завдяки cgroups, звільняє розробників від необхідності запускати кожен мікросервіс у власній віртуальній машині, що в свою чергу звільняє обчислювальну потужність, раніше виділену для цих віртуальних машин, і пропонує її розміщеним додаткам.

Деякі додатки вимагають значної обчислювальної потужності, тоді як інші генерують багато мережевого трафіку. При розміщенні мікросервісів необхідно правильним чином разраховувати сумарне використання ресурсів сервера – розробники можуть максимізувати рівні використання всіх ресурсів сервера, а не просто розмістити на ньому декілька додатків, залежних від процесорного часу, ніяк не використовуючи при цьому мережевий ресурс чи пам’ять. Інженери Google автоматизували даний процес розміщення у своїй системі Borg, частину якої зробили публічною та безкоштовною – Kubernetes [15].

1.3 Аналіз архітектури Kubernetes

В даному розділі аналізується архітектура кластеру Kubernetes, а також особливості моніторингу та управління ресурсами, алгоритм роботи планувальника. Крім цього розглядається сервер метрик та його API, а також систему моніторингу Prometheus. Опис загальної архітектури Kubernetes не наводиться, оскільки в даній

роботі важливо розуміти алгоритм роботи Kubernetes саме в контексті моніторингу та управління обчислювальними ресурсами.

1.3.1 Обчислювальні ресурси в кластері Kubernetes

Керування ресурсами кластеру Kubernetes – це важлива складова підтримки кластеру, яка може, при правильному керуванні, значно покращити продуктивність та фінансову складову. Якщо не контролювати використання мікросервісами обчислювальних ресурсів, то вони можуть значно заважати один одному – конкурувати за процесорний час та пам’ять, що призведе до погіршення їх продуктивності. Якщо задати квоти для мікросервісу занадто високі, то Kubernetes може почати штучно сповільнювати центральний процесор або навіть аварійно завершувати поди з помилкою OOM. З іншого боку, якщо поду задати занадто багато, то ресурси кластеру швидко закінчаться, або не буде доступних вузлів, що задовільняють вимоги по ресурсам, і Kubernetes не зможе розмістити под. На жаль, планування використання обчислювальних ресурсів у кластері Kubernetes є досить складною задачею.

Kubernetes має всього два стандартні керовані ресурси: процесорний час та пам’ять. Одиницями центрального процесора є ядра, а пам'ять задається байтах. Ці два ресурси відіграють важливу роль у тому, як планувальник розміщує поди на вузлах.

Коли описується под, є можливість додатково вказати, скільки кожного з ресурсів потрібно контейнеру для роботи. Для цього існують спеціальні поля в специфікації поду – це request (запит ресурсів) та limit (обмеження ресурсів). Вони можуть не вказуватися, тоді планувальник не буде враховувати ресурсну складову при розміщенні на вузлі. Або можуть як разом, так і окремо. Коли вказується запит на ресурси для контейнеру в поді, планувальник використовує цю інформацію, щоб вирішити, на якому вузлі розмістити под. Коли вказується обмеження на ресурси для контейнера, kubelet слідкує за тим, щоб под не використовував більше цього ресурсу,

ніж встановлене розробником обмеження. Kubelet також резервує ресурси по запиту для використання лише цим подом.

Якщо на вузлі, на якому запущено под, є достатньо необхідного ресурсу, то контейнеру дозволено використовувати більше ресурсів, ніж вказано в запиті. Однак контейнеру заборонено використовувати більше, ніж вказано в обмеженнях. Наприклад, якщо встановити запит пам’яті в розмірі 256 Мб, і цей под потрапить на вузол, на якому є вільних 8 ГБ пам’яті, тоді контейнер може використовувати всю доступну оперативну пам’ять. Якщо для цього контейнера встановлено ліміт по пам’яті в 4 ГБ, kubelet не дозволить запросити більше ніж 4 ГБ.

Наприклад: коли процес у контейнері намагається зайняти більше пам'яті, ніж вказано в лімітах, то kubelet завершує процес з помилкою нестачі пам'яті (OOM). Обмеження можуть бути реалізовані як реактивно (kubelet втручається, коли виявляє перевищення), так і примусово (за допомогою linux cgroup). Різні середовища можуть по-різному застосовувати однакові обмеження.

Якщо в специфікації поду вказаний ліміт на пам’ять, але не вказано запит, то Kubernetes автоматично призначає запит на пам’ять еквівалентний ліміту. Подібним чином, якщо для поду вказано ліміт на використання процесора, але не визначено запит, то Kubernetes автоматично призначає запит по процесору, який відповідає ліміту.

Далі розглядаються основні типи ресурсів. Процесор та пам’ять – це основні ресурси, які можуть конфігуруватися. Кожен тип ресурсу має базові одиниці. Ресурс процесора відповідає за обчислювальну потужність і вказується в спеціальних одиницях процесорів. Пам'ять вказується в байтах. В версії Kubernetes v1.14 додали новий ресурс – hugepages. Це здатність ОС Linux виділяти значно більші сторінки пам’яті, чим стандартні в системі 4 кілобайти [16].

За допомогою блоку pod.containers.resources можна задавати ліміти та запити на обчислювальні ресурси в Kubernetes. Доступні такі поля в специфікації подів:

- pod.containers.resources.limits.cpu;
- pod.containers.resources.limits.memory;
- pod.containers.resources.limits.hugepages;

- pod.containers.resources.requests.cpu;
- pod.containers.resources.requests.memory;
- pod.containers.resources.requests.hugepages.

Перші три поля відповідають за ліміти використання, а останні три – за запит ресурсів. Різницю між лімітом та запитом вже розглянуто в цьому розділі. Ліміти та запити на ресурс процесора вимірюються в одиницях процесора Kubernetes. Один процесор Kubernetes еквівалентний 1 віртуальному ядру для хмарних провайдерів та 1 потоку(з урахуванням технології HyperThreading) на bare-metal процесорах [17]. Дозволено вказувати дробові значення для запитів та лімітів. При специфікації з resources.requests.cpu 0.5 гарантується вдвічі менше процесорного часу, ніж тому контейнеру зі значенням 1 ЦП. Значення 0.1 еквівалентно значенню 100m, де m – millicore. Kubernetes автоматично трансформує дробові значення в значення в мілікорах. Точність, менша за 1m, не допускається. Значення 0.1 – це завжди однакова кількість процесорного ресурсу, незалежно від кількості ядер процесору – як на одноядерній, так і на 48–ядерній машині [18].

Ліміти та запити на пам'ять вимірюються в байтах. Значення можуть задаватися як цілі і дробові числа з використанням суфіксів: Ti – терабайт, Gi – гігабайт, Mi – мегабайт та Ki – кілобайт.

В специфікації, зображеній на рисунку 1.7, маємо еквівалентні ліміти та запити, хоча строкові значення різні:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "0.256Gi"
    cpu: "2.5"
```

Рисунок 1.7 – Приклад специфікації запитів та лімітів на ресурси

Проаналізуємо класи подів в Kubernetes. У випадках, коли вузол повністю витрачає обчислювальні ресурси – сума всіх обчислювальних ресурсів запущених

контейнерів перевищує обчислювальний ресурс самого вузла. У таких екстремальних випадках Kubernetes починає видаляти запущені контейнери на вузлі та переносити їх на інші. Спочатку необхідно видалити менш критичні контейнери, щоб звільнити ресурси для більш важливих додатків. Яким чином розробники можуть вказати критичність того чи іншого поду або контейнера. За допомогою класів QoS [19]:

- best-effort;
- guaranteed;
- burstable.

Best-effort клас означає, що поди без вказаних запитів та лімітів на ресурси припиняються першими у разі вичерпання ресурсів.

```
resources:
  requests:
  limits:
```

Рисунок 1.8 – Приклад специфікації для best-effort класу

Guaranteed клас призначається подам, де вказані запит та ліміт для всіх ресурсів і при цьому ліміти і запити еквівалентні. Ці поди мають високий пріоритет, тому вони видаляються, лише якщо перевищуються ліміти та немає подів нижчого пріоритету – best-effort.

```
resources:
  requests.memory: "256Mi"
  requests.cpu: "250m"
  limits.memory: "256Gi"
  limits.cpu: "250m"
```

Рисунок 1.9 – Приклад специфікації для guaranteed класу

Burstable клас призначається подам, у яких вказані обмеження та запити для обох ресурсів – пам’яті та процесора і їх значення не однакові. Поди цього класу мають найвищий пріоритет та видаляються з вузла коли немає Guaranteed або best-effort подів.

```
resources:
  requests.memory: "512Mi"
  requests.cpu: "250m"
  limits.memory: "512Mi"
  limits.cpu: "500m"
```

Рисунок 1.10 – Приклад специфікації для burstable класу

Оскільки час безвідмовної роботи мікросервісів є частиною SLA, що дуже важливо для бізнесу, то необхідно застосувати правильні ліміти та запити на ресурси, що надасть можливість правильно розподілити ресурси для критичних програм, а QoS допоможе контролювати життєвий цикл програми у випадку вичерпання ресурсів на вузлі.

За замовчуванням в кластері Kubernetes кількість доступних ресурсів для контейнерів не обмежується. За допомогою спеціальних API ресурсів адміністратори кластера можуть задати стандартні для всіх подів та контейнерів значення на ліміти та запити в просторі імен, таким чином, якщо у специфікації поду не буде вказано квот ресурси, то Kubernetes автоматично застосує стандартні значення. Використання такого інструменту вважається хорошою практикою. Одним із таких інструментів для задання стандартних квот для кластера є LimitRange [20].

LimitRange надає обмеження, які можуть:

- застосовує мінімальне та максимальне використання обчислювальних ресурсів на поді або контейнері у разі відсутності квот у специфікації поду;
- конфігурування співвідношення між запитом і лімітом у разі відсутності одного з них;
- гарантування того, що задані квоти не перевищують вказаних системним адміністратором при конфігуруванні LimitRange ресурсу. Наприклад, в специфікації поду вказано, що контейнеру необхідно 10 Gi пам’яті, проте в LimitRange максимальне значення пам’яті – 5 Gi; в такому випадку под не буде розміщений взагалі;

Приклад створення політики для простору імен за допомогою LimitRange на рисунку 1.11:

```
limits:
- default:
    cpu: 800m
  defaultRequest:
    cpu: 800m
  max:
    cpu: 800m
  min:
    cpu: 200m
  type: Container
```

Рисунок 1.11 – Специфікація для LimitRange

Поле default та defaultRequest відповідають за стандартні значення, які будуть застосовані у випадку відсутності квот у специфікації поду. А поля max та min – що вказані квоти не перевищують максимального та мінімального значення.

1.3.2 Моніторинг використання ресурсів

Моніторинг в кластері Kubernetes важливий у контексті використання ресурсів та контролю фінансових витрат. Кластером Kubernetes потрібно активно проводити моніторинг та керувати, щоб забезпечити ефективне використання обчислювальних ресурсів вузлів всіма підсистемами.

Моніторинг в Kubernetes можна розділити на дві основні сфери.

Моніторинг системних метрик, що складається з метрик kubelet, оцінювач ресурсів (resource estimator), metrics-server і API server, що обслуговує API несистемних метрик [21]. Ці метрики використовуються основними системними компонентами, такими як планувальник, контролери горизонтального та вертикального масштабування та деякі компоненти інтерфейсу адміністратора кластеру (наприклад, kubectl top). Дана частина метрик не призначена для інтеграції зі сторонніми системами моніторингу [10].

Моніторинг загальних метрик, що використовується для збору різних показників із системи та додатків та надання їх розробникам, а також контролерам горизонтального та вертикального масштабування через метрики мікросервісів.

Kubernetes поставляється без систем для збору даних метрик, але існує багато готових до використання рішень – Prometheus, Heapster і так далі.

В даній магістерській дисертації розглядається саме перший тип метрик, оскільки включає метрики використання ресурсів мікросервісами.

В кластері Kubernetes є два основні джерела системних метрик:

- kubelet, що надає інформацію про використання ресурсів подами та контейнерами на кожному с вузлів;
- оцінювач ресурсів, який отримує метрики з kubelet, обробляє їх, агрегує та зберігає для подальшого використання.

Обидва джерела постійно опитується сервером метрик, який обробляє і зберігає показники. Сервер метрик зберігає лише останні значення і не має постійного сховища, тому ці метрики необхідно експортувати в сторонню базу даних. Для того, щоб отримати поточні показники з сервера метрик існує спеціальне API, яким і користуються системні компоненти кластеру, а також для експортування.

Процес збору та обробки системних метрик реалізований дуже просто, тому всі компоненти можливо замінити на власні. Так, наприклад, є можливість реалізувати сервер метрик, який зберігає не лише останні значення, а за деякий період, або замінити оцінювач ресурсів чи планувальник, якщо розробника не влаштовують вбудовані методи оцінки.

Сервер метрик – це базовий інструмент, на якому працюють kubectl top, kube-dashboard, планувальник та високорівневі системи моніторингу, такі як Prometheus та Heapster [22]. Сервер метрик – це джерело системних метрик для внутрішнього використання в Kubernetes, а також для автоматичного масштабування – автоматичного горизонтального масштабування на основі системних метрик та вертикального масштабування. Сервер метрик збирає значення лише для ресурсів, необхідних для автоматичного масштабування: процесор і пам’ять. Значення метрик використовують стандартні одиниці – мілікори та кілобайти для процесора та пам’яті відповідно. Значення на сервері обновляться приблизно з інтервалом в 15 секунд та є середнім значенням за цей часовий проміжок.

1.3.3 Система моніторингу Prometheus

Серед основних інструментів для моніторингу найвідомішим рішенням є Prometheus – це набір інструментів з відкритим кодом для моніторингу, розроблений спеціально для Kubernetes. У попередньому підрозділі розглядаються типи метрик, проте пропущено опис поняття метрики. Метрики – це значення прив’язані до часу, контекст яких ігнорується, а важливим є саме значення.

Прикладами метрик, які є у кожного проекту, є кількість оброблених HTTP запитів, час витрачений на обробку запитів і скільки запитів обробляється на даний момент. Можуть бути і складніші варіанти, наприклад, для HTTP-запиту можна порахувати кількість входжень конкретного URL. Розробники можуть використовувати метрики для відстеження затримки та обсягів даних, що обробляються кожною з підсистем у додатках, полегшуючи відлагодження проблем з продуктивністю. Таким же чином, метрики використовуються для моніторингу використання ресурсів, а базовою метрикою є значення поточного використання деякого ресурсу та час зняття метрики.

Prometheus часто обирається для системних метрик, оскільки вирішує проблему зберігання історичних даних. Для запиту метрик існує спеціальна мова PromQL, яка є дуже потужною. Більш того метрики з Prometheus легко візуалізувати за допомогою спеціальних інструментів, наприклад, Grafana [23]. На рисунку 1.12 зображена архітектура даного рішення.

На рисунку 1.12 зображена архітектура Prometheus. Prometheus постійно опитує спеціальні сервери – експортери, які в свою чергу опитують прив’язані мікросервіси по протоколу HTTP. У кожен мікросервіс вбудований спеціальний клієнт, який запускає сервер в окремому поточу, та який збирає необхідні метрики та зберігає їх в пам’яті, чекаючи поки експортер їх збере.

Основні компоненти діаграми:

- Prometheus server – основна програма, яка збирає метрики з експортерів та Pushgateway;

- Pushgateway – спеціальний сервер, який може отримувати метрики, що необхідно для програм, які можуть завершитись швидше, ніж їх опитає експортер;
- Alertmanager – компонент, який відповідає за оповіщення;
- Exporters – програми, які постійно опитують мікросервіси для отримання метрик, після чого відправляють їх на основний сервіс.
- Service discovery – Prometheus постійно шукає нові поди мікросервісів для отримання метрик і, якщо, адреса поду змінилася, то Prometheus автоматично збере метрики з нової адреси.

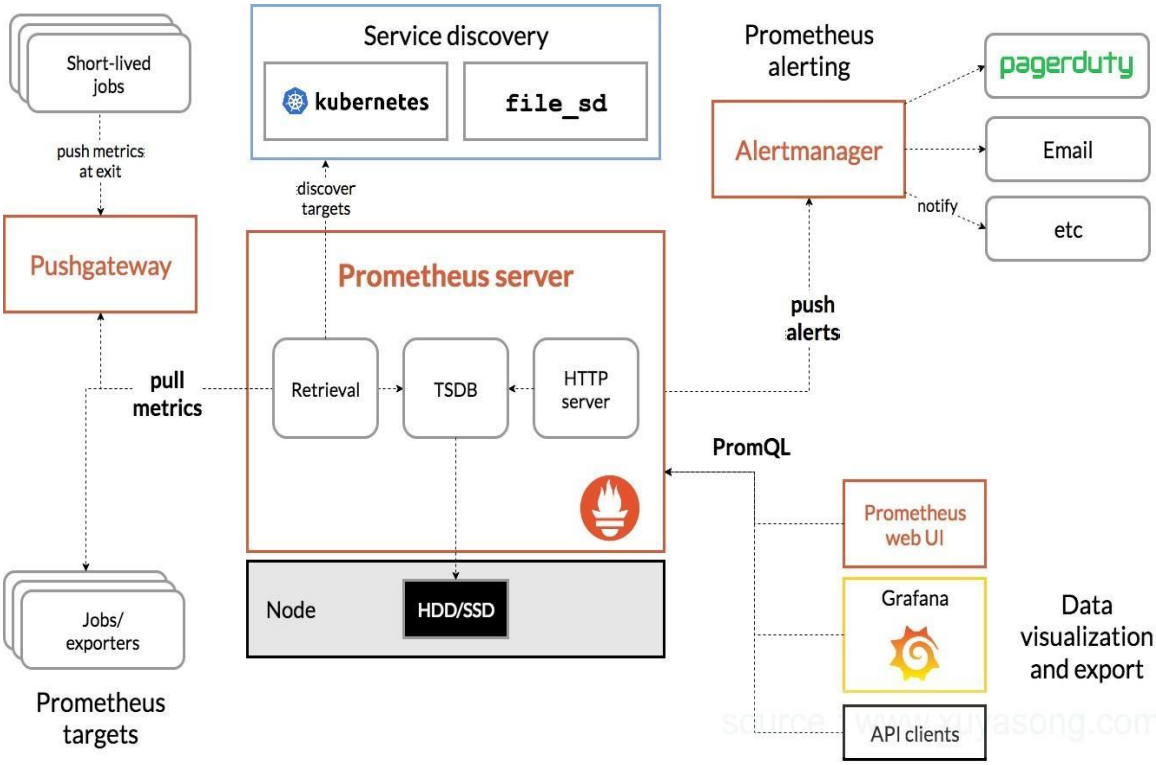

Рисунок 1.12 – Архітектура Prometheus [24]

Алгоритм роботи виглядає так:

1. Сервер Prometheus регулярно отримує метрики з експортерів, або збирає з Pushgateway та з інших серверів Prometheus.
2. Сервер Prometheus зберігає зібрані метрики локально, агрегує та перевіряє правила Alertmanager, після чого надсилає попередження до Alertmanager.
3. Alertmanager обробляє отримані сповіщення та відправляє у необхідний канал зв’язку.

Отже, для того, щоб Prometheus дізнався про мікросервіс та зібрал метрики додатку необхідно мати клієнт або надсилати метрики до Pushgateway, проте системні метрики надсилаються до сервера метрик, а він такого клієнта не має. Для того, щоб доставити системні метрики в Prometheus, існують спеціальні експортери, які опитують сервер метрик через вже розглянуте API, приводять до необхідного формату та надсилають на основний сервер. Дана схема зображена на рисунку 1.13.

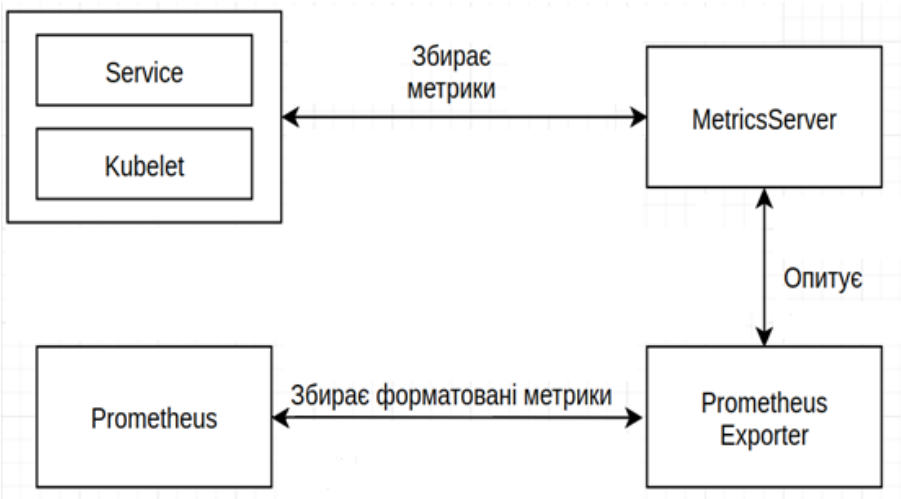

Рисунок 1.13 – Алгоритм збору системних метрик

1.4 Аналіз існуючих рішень

1.4.1 Аналіз Vertical Pod Autoscaler

Vertical Pod Autoscaler, що можна знайти в публічному репозиторії https://github.com/kubernetes/autoscaler, має дві функції: автоматизація процесу резервування ресурсів для подів (якщо доступних ресурсів не вистачає) та оптимізація використання кластерних ресурсів.

Vertical Pod Autoscaler проводить моніторинг використання обчислювальних ресурсів в кластері та відстежує події аварійних завершень додатків та пропонує оптимальні квоти ресурсів для окремих подів, що можливо застосувати до всіх подів додатку.

Для сбору метрик використання ресурсів використовується вбудований в Kubernetes сервіс metrics-server, який надає дані про використання додатками пам’яті та CPU в реальному часі.

Серед недоліків даного рішення можна зазначити:

- оновлення квот на ресурси від VPA призводить до перезапуску всіх запущених контейнерів;
- в даний час VPA не можна одночасно використовувати з горизонтальним автомасштабуванням на основі використання ресурсів CPU чи пам'яті;
- Vertical Pod Autoscaler не працює з додатками на основі JVM через обмежену видимість фактичного використання пам'яті;
- Vertical Pod Autoscaler може працювати лише в автоматичному режимі.

Vertical Pod Autoscaler включає два основні компоненти для роботи автомасштабування – Recommender та Updater. Recommender VPA збирає метрики використання ресурсів та перевіряє поточні квоти, після чого рекомендує оптимальні значення для квот. Updater на основі рекондомендій задає нові квоти для подів та при необхідності переносить запущений под на іншу машину. VPA задаватиме квоти в межах зазначених адміністратором значень [28]. Приклад:

```yaml
apiVersion: autoscaling.k8s.io/v1beta2
kind: VerticalPodAutoscaler
metadata:
  name: nginx-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment
    name:       nginx
  resourcePolicy:
    containerPolicies:
    - containerName: "nginx"
      minAllowed:
        cpu: "250m"
        memory: "100Mi"
      maxAllowed:
        cpu: "2000m"
        memory: "2048Mi"
```

Рисунок 1.14 – Приклад конфігурації для VPA

На рисунку 1.14 зображено конфігурацію для Vertical Pod Autoscaler, якій apiVersion та kind описують компонент та його версію, а саме VerticalPodAutoscaler версії v1beta2. В блоці spec.targetRef описано цільовий сервіс для моніторингу та конфігурування, де kind – тип об’єкту, який відстежується, name – назва додатку, який є цільовим. Блок resourcePolicy відповідає за задання мінімальних та максимальних значень для квот для кожного конкретного контейнеру.

Vertical Pod Autoscaler (VPA) наразі не є загальнодоступним рішенням та знаходиться в режимі тестування. VPA – це розширення для кластеру Kubernetes, яке дозволяє автоматично встановлювати значення для ресурсів процесора та пам'яті контейнера. VPA працювати в автоматичному режимі або в напівавтоматичному – встановлювати автоматично значення для запитів та обмежень ресурсів процесора та пам'яті, або рекомендувати дані значення.

VPA збирає метрики використання контейнером CPU та RAM та на основі цих даних вираховує необхідні обмеження ресурсів, що зображено на рисунку 1.15, де VPA CRD – екземпляр VPA, dev – адміністратор, який конфігурує обмеження ресурсів, pod – екземпляр контейнеризованого додатку.

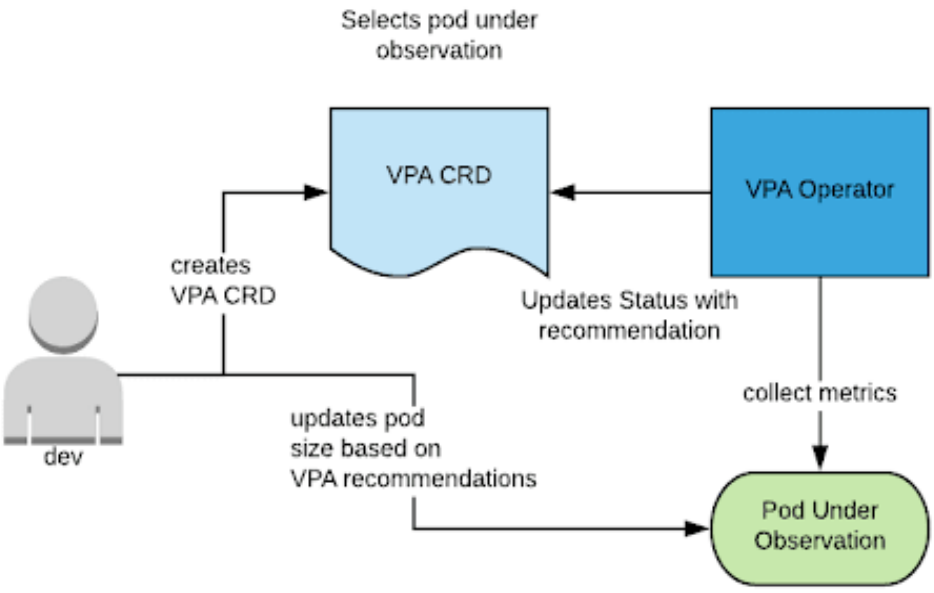

Рисунок 1.15 – Діаграма роботи VPA [28]

Recommender – основний компонент VPA. Він відповідає за обчислення рекомендованих запитів на ресурси. На початку своєї роботи даний компонент завантажує історію використання ресурсів всіх подів, незалежно від того, чи використовують вони VPA, разом з історією подій OOM з бази даних. Ці дані агрегуються та зберігаються в пам’яті для підвищення продуктивності.

Під час роботи recommender постійно збирає нові метрики використання та події OOM використовуючи API для отримання подій кластеру та сервер метрик, розглянутий в розділі 1.4.3. Крім того, даний компонент спостерігає за всіма подами та об'єктами VPA в кластері. Для кожного поду, якому відповідає один із міток VPA, recommender обчислює рекомендовані ресурси.

VPA Admission Controller перехоплює всі запити на створення подів. Якщо перехоплений под відповідає конфігурації VPA з необхідним режимом, то контролер переписує запит, застосовуючи рекомендовані ресурси для специфікації поду. В іншому випадку специфікація поду залишається незмінною. Даний контролер отримує рекомендовані ресурси від іншого компоненту – Recommender.

Updater – це компонент, відповідальний за застосування рекомендованих значень запитів та лімітів на ресурси до специфікацій подів та розгортань. Він відстежує всі об'єкти VPA та мікросервіси в кластері, періодично отримуючи рекомендації для них, викликаючи API іншого компоненту VPA – Recommender. Коли рекомендовані ресурси суттєво відрізняються від поточних квот на ресурси, даний компонент може ініціювати перерозгортання подів з новими значеннями квот. Незважаючи на те, що припинення роботи поду для зміни лімітів на ресурси є небажаним та може призвести до помилок в роботі мікросервіса, це іноді виправдано для того, щоб уникнути тротлінгу процесора та зменшити ризик аварійного завершення через OOM або заощадити ресурси протягом тривалого періоду часу.

VPA аналізує та задає оптимізовані квоти лише для подів зі встановленим значення updatePolicy.mode. Updater компоненту при задаванні нових квот для поду враховує загальний стан кластеру для того, щоб впевнитись в достатній кількості ресурсів з новими квотами.

VPA контролює запити на пам'ять і ресурс процесора контейнерів. Запит обчислюється на основі аналізу поточних та історичних значень використання ресурсів контейнера. Рекомендаційна модель передбачає, що споживання пам'яті та центрального процесора є незалежними випадковими величинами з розподілом, рівним тому, який спостерігався за останні N днів (рекомендованим значенням є N = 8 для фіксації тижневих піків). В майбутньому VPA зможе виявляти тенденції, періодичність та інші закономірності.

Для ресурсу процесора метою є утримувати частку часу, коли використання контейнера перевищує високий відсоток (наприклад, 95%) запиту, нижче певного порогу (наприклад, 1% часу) [27]. У даній моделі використання центрального процесора визначається як усереднене використання за деякий короткий інтервал. Чим коротший інтервал вимірювання, тим оптимальніші рекомендації будуть розраховуватися. Мінімальна роздільна здатність для отримання рекомендацій – раз на хвилину, рекомендована – щосекунди.

Для пам'яті метою є утримування ймовірності перевищення подом ліміту за певний часовий проміжок нижче певного порогу (наприклад, нижче 1% за 24 години). Цей інтервал повинен бути достатньо довгим (≥ 24 год), щоб перезапуски подів, спричинені OOM, суттєво не впливали на роботу мікросервісів.

Коли контейнер зупиняється через перевищення лімітів пам'яті, його фактичні вимоги до пам'яті невідомі – останні отримані метрики, очевидно, є меншими за актуальні. Дана ситуація вирішується спеціальним коефіцієнтом “запас міцності” для останнього отриманого значення споживання пам’яті.

На рисунку 1.16 зображено приклад роботи VPA на прикладі графіку. На даному графіку верхня крива – це ліміт по пам’яті для деякого поду, а нижня крива – поточне використання пам’яті цим подом. Бачимо, що на початку графіку ліміт був заданий в 4.5 Гб при фактичному споживанні не більше 2 Гб. VPA о 6:00 на графіку поетапно зменшив ліміт на ресурси для поду. На даному графіку видно, що крива поточного використання в деяких місцях відсутня, що сталося через завершення роботи мікросервісу с OOM помилкою.

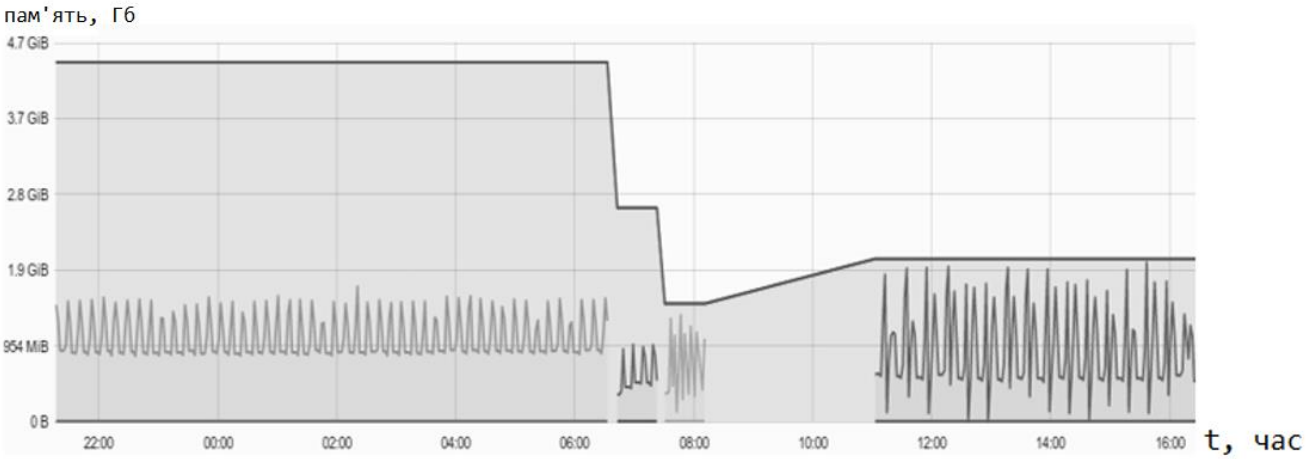

Рисунок 1.16 – Приклад роботи VPA при регулюванні пам’яті

Після того, як VPA відстежив OOM помилку, то встановив значення ліміту пам’яті вище, після чого мікросервіс відновив роботу. Отже, VPA в даному випадку звільнив приблизно 2.5 Гб оперативної пам’яті для використання іншими мікросервісами, але його робота призвела до завершень програми через помилку OOM.

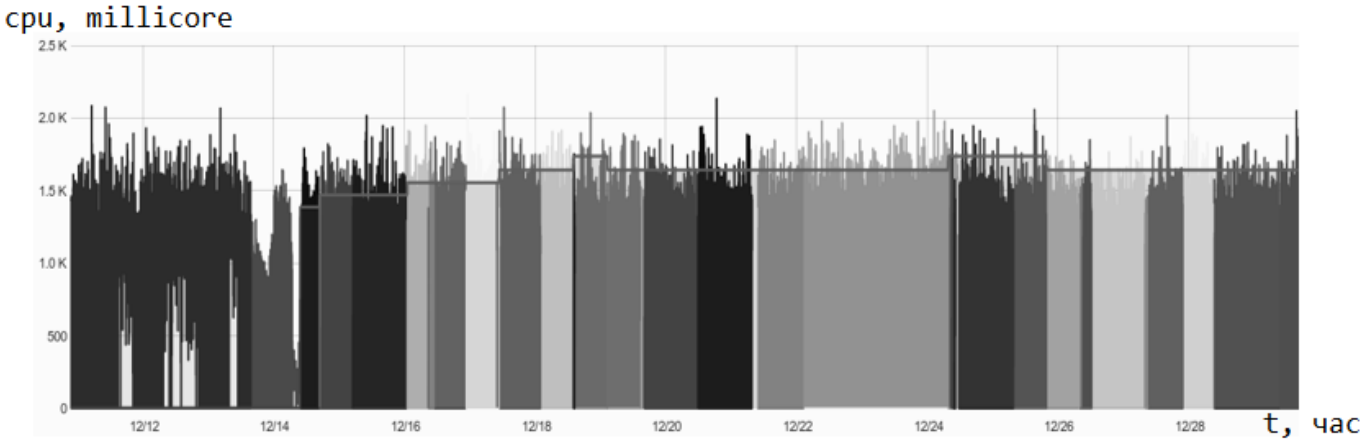

Рисунок 1.17 – Приклад роботи VPA при регулюванні ресурсу процесора

На рисунку 1.17 бачимо аналогічний графік, але для ресурсу процесора. На графіку можна спостерігати, що значення використання часто є вищими за встановлений ліміт. Це нормальна практика для Kubernetes, оскільки ресурс процесору не такий критичний, як оперативна пам’яті. Якщо споживання цього ресурсу перевищить ліміт, то kubelet штучно буде сповільнювати програму, що не

призведе до її завершення. Тому ліміт для процесора можна тримати рівним поточному споживанню.

Далі розглядається у якому вигляді та де зберігаються рекомендації VPA. Для отримання цих даних необхідно виконати команду на вашому кластері kubectl describe vpa POD_NAME, де POD_NAME – унікальна назва розгортання. Результат виконання зображений на рисунку 1.18.

На рисунку 1.18 маємо декілька рекомендованих значень:

- поле Lower Bound містить нижню границю, і, якщо фактичне значення менше, то автоматично ліміт для даного поду буде зменшений, а сам перезапущений з новими квотами;
- поле Upper Bound містить верхню границю, при перевищенні якої поду автоматично буде виділено додаткові ресурси, а також под перезапуститься;
- поле Uncapped Target містить ліміт, який є оптимальним, та який встановиться без врахування максимального та мінімального обмеження у специфікації VPA;
- поле Target містить значення, яке буде встановлено під час наступного запиту на створення поду.

```
Recommendation:
  Container Recommendations:
    Container Name:  hamster
    Lower Bound:
      Cpu:     519m
      Memory:  262144k
    Target:
      Cpu:     587m
      Memory:  262144k
    Uncapped Target:
      Cpu:     587m
      Memory:  262144k
    Upper Bound:
      Cpu:     1
      Memory:  500Mi
```

Рисунок 1.18 – Рекомендації VPA

1.4.2 Аналіз Magalix KubeOptimizer

Рішення Magalix KubeOptimizer виконує лише одну функцію – це моніторинг та рекомендації оптимальних квот для ресурсів [33]. Magalix постійно контролює розподіл та використання процесора та пам'яті для кожного поду.

Дане рішення аналізує схеми та патерни використання ресурсів та використовує статистичні методи, щоб оцінити необхідні квоти для CPU та пам'яті. Рекомендовані значення можна отримати на спеціальному дашборді, а також одразу їх застосувати. Серед переваг – відстежування патернів використання та застосування статистичних методів, відстежування несправностей в кластері. Основні недоліки – дане рішення є платним, відсутність автоматичного режиму роботу.

На відміну від VPA, має потужний графічний інтерфейс, а також сповіщає про проблеми з ресурсами або помилками OOM.

ISSUE	ADVISOR	ENTITY TYPE	LAST SCAN
Containers wasting memory resources	Container Resources Advisor	container	10 hours ago
Container missing CPU limit or request	Container Resources Advisor	container	10 hours ago
Containers starving in CPU can be throttled	Container Resources Advisor	container	10 hours ago
Containers starving in memory and can crash	Container Resources Advisor	container	10 hours ago
Cost Saving	Node Advisor	cluster	11 hours ago
Container having high utilization CPU	Container Resources Advisor	container	10 hours ago
Better margins / Performance	Node Advisor	cluster	11 hours ago

Рисунок 1.19 – Перелік несправностей Magalix

Далі розглядається графічна панель моніторингу. Одна із особливостей даного рішення – це постійний моніторинг кластеру на предмет проблем, наприклад, відсутні квоти на ресурси, виділено занадто багато пам’яті для поду або занадто низька утилізація кластеру. Приклад на рисунку 1.19. Також можна отримати рекомендовані квоти на ресурси для кожного мікросервісу, як на прикладі 1.20:

	Limits	Request
	CPU	CPU
Current	1 core	500 millicores
Recommended	1 core	50 millicores

Рисунок 1.20 – Рекомендації по квотам на Magalix

Є можливість встановити рекомендовані метрики прямо з графічної панелі, що зображено на рисунку 1.21:

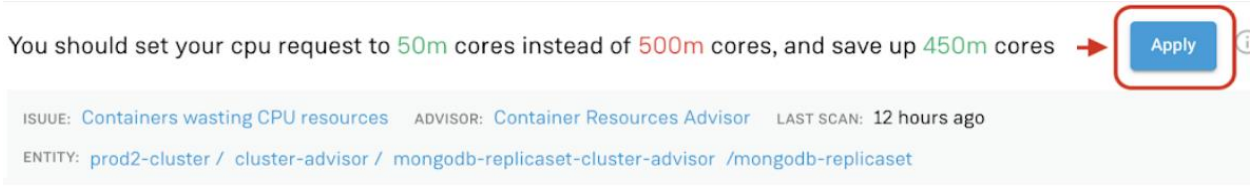

Рисунок 1.21 – Встановлення рекомендованих ресурсів через Magalix

Також Magalix проводить моніторинг всіх мікросервісів та візуалізує всі необхідні метрики – використання пам’яті та процесора, кількість подій OOM у кластері, тротлінг мікросервісів.

Також можна оцінити ваш кластер та отримати інформацію по утилізації, кількості вузлів, використанню пам’яті та процесора, а також постійної пам’яті, що зображено на рисунку 1.22.

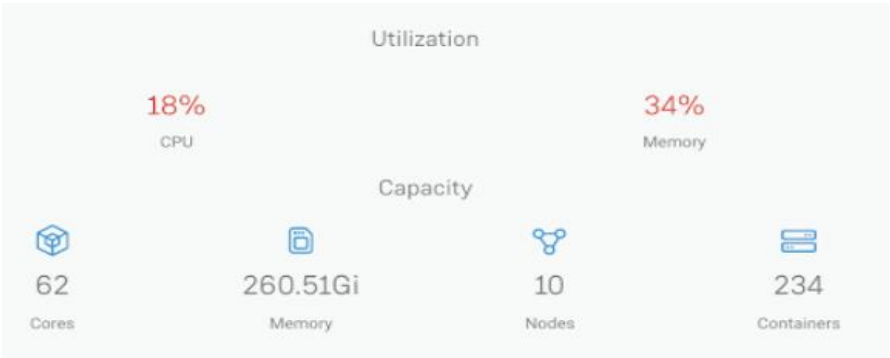

Рисунок 1.22 – Оцінка кластеру на Magalix

1.4.3 Аналіз Goldilocks

В основі рішення використовується Vertical Pod Autoscaler. Основна відмінність заключається в тому, що при роботі даного рішення VPA працює в режимі рекомендацій.

Для того, щоб надати рекомендації, Goldilocks використовує програмне рішення, що розглядається в розділі 1.4.1 (VPA). Архітектура контролера VPA містить механізм рекомендацій, який враховує поточне використання ресурсів ваших подів для генерування оптимальних значень квот. Основною функція VPA є встановити оптимальні значення автоматично, але якщо ваша мета просто отримати ці значення для подальшого аналізу, то VPA не дуже підходить. Наприклад, горизонтальний контролер автоматичного масштабування – HPA – не може працювати разом з VPA, тому доводиться просто використовувати механізм рекомендацій, який надає VPA, для отримання оптимальних запитів та обмежень ресурсів і їх ручного застосування [30].

Дане рішення аналізує кожен об'єкт розгортання та створюємо відповідний об'єкт VPA. Для VPA встановлюється режим, в якому відбувається лише моніторинг та запис оптимальних значень в базу даних, але не встановлення цих значень в специфікаціях розгортань. Для того, щоб переглянути ці рекомендації, потрібно для кожного розгортання використовувати kubectl для запиту рекомендованих значень VPA. При значному об’ємі подів в розгортаннях, а також великої кількості розгортань, це може бути кропіткою ручною роботою.

Інформаційна панель Goldilocks забезпечує візуалізацію рекомендацій VPA. Дана панель стає доступною у кластері після встановлення додатку. Є можливість переглянути два типи рекомендацій, залежно від того, який клас QoS є бажаним для розгортань.

Guaranteed клас призначається подам, де вказані запит та ліміт для всіх ресурсів і при цьому ліміти і запити еквівалентні. Ці поди мають високий пріоритет, тому вони видаляються, лише якщо перевищуються ліміти та немає подів нижчого пріоритету – best-effort.

Burstable клас призначається подам, у яких вказані обмеження та запити для обох ресурсів – пам’яті та процесора і їх значення не однакові. Поди цього класу мають найвищий пріоритет та видаляються з вузла коли немає Guaranteed або best-effort подів.

На рисунку 1.23 зображено графічний інтерфейс Goldilocks для одного з розгортань. В цьому графічному інтерфейсі можна встановити ліміти та запити для кожного контейнеру даного розгортання. Режими «Guaranteed QoS» та «Burstable QoS» дозволяють застосувати необхідні класи обслуговування, які описані вище у цьому розділі. Також на даній панелі можна побачити поточні значення лімітів на запитів на ресурси та порівняти з рекомендованими.

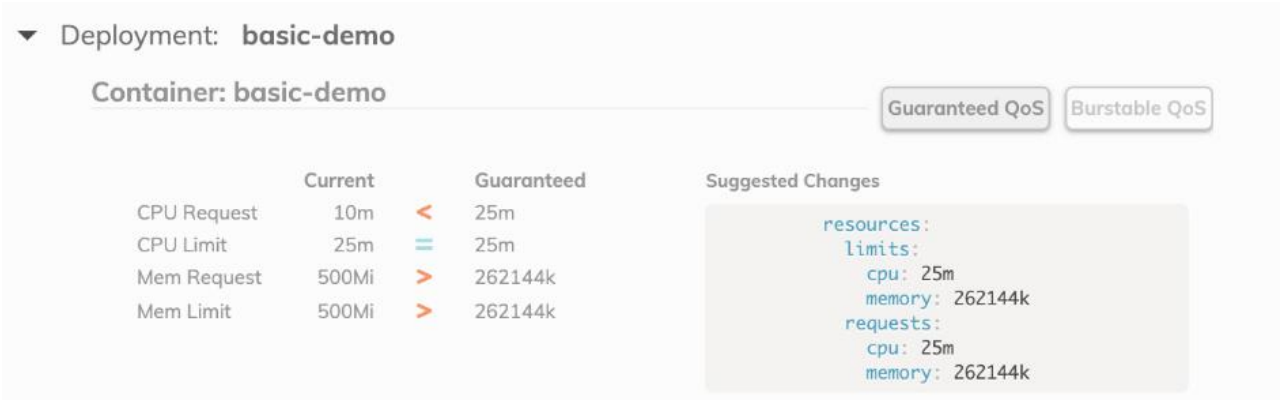

Рисунок 1.23 – Приклад інформаційної панелі Goldilocks

Ідея даного рішення є дуже простою, але дієвою, оскільки дозволяє автоматизувати процес керування ресурсами без конфігурування VPA, що є складною задачею.

1.5 Висновки

В даному розділі розглядаються основні поняття при розробці систем для моніторингу та керування обчислювальними ресурсами в кластері Kubernetes.

Результати проведеного аналізу архітектури Kubernetes та існуючих рішень викостовуються при розробці структурної схема та програмної реалізації системи.

По-перше, на початку даного розділу розглянуто мікросервісну архітектуру, її основні переваги та недоліки, проведено порівняння з монолітною архітектурою, наведені приклади мікросервісної архітектури. В підрозділі 1.2 коротко описано, чому в мікросервісній архітектурі необхідна контейнеризація і її переваги перед віртуалізацією.

По-друге, в розділі 1.3 розглядаються обчислювальні ресурси кластеру, а саме пам’ять та процесорний час, а також методи керування ними – задання лімітів та запитів на ресурси в специфікації подів та Kubernetes ресурс LimitRange, який дозволяє обмежувати дані ліміти та ресурси, а також задавати стандартні значення для простору імен, що можна використати для обмеження автоматично встановлених квот.

Також в підрозділі 1.4 розглядаються методи моніторингу використання ресурсів у кластері. Основним джерелом таких метрик убло обрано сервер метрик, який використовується у внутрішніх процесах Kubernetes, а також для роботи горизонтальних та вертикальних контролерів масштабування, який використано під час реалізації програмного рішення. В даному розділі досліджено API серверу метрик, а також розглянуто систему моніторингу Prometheus.

Крім того, розглядаються алгоритм роботи планувальника Kubernetes та фактори, які впливають на розміщення подів, оскільки це важливо в контексті динамічної зміни квот на ресурси. З’ясовано, що змінювати запити та ліміти на ресурси під час роботи вже розміщеного поду не має сенсу, оскільки kubelet отримує конфігурацію тільки при запуску поду, але не під час роботи. Отже, щоб змінити квоти на ресурси, необхідно под перезапускати.

В даному розділі розглянуто існуючі рішення, а саме:

- Vertical Pod Autoscaler;
- Magalix KubeOptimizer;
- Goldilocks.

Vertical Pod Autoscaler має дві функції: автоматизація процесу резервування ресурсів для подів (якщо доступних ресурсів не вистачає) та оптимізація використання кластерних ресурсів. VPA проводить моніторинг використання

обчислювальних ресурсів в кластері та відстежує події аварійних завершень додатків та пропонує оптимальні квоти ресурсів для окремих подів, що можливо застосувати до всіх подів додатку.

Goldilocks базується на VPA, автоматично створює необхідні для роботи VPA об’єкти та пропонує графічний інтерфейс.

Magalix KubeOptimizer – платне комплексне рішення, яке також включає моніторинг та керування ресурсами. На відміну від VPA, має потужний графічний інтерфейс, а також сповіщає про проблеми з ресурсами або помилками OOM.

2 РОЗРОБЛЕННЯ СТРУКТУРНОЇ СХЕМИ СИСТЕМИ

В даному розділі розробляється структурна схема системи, а також аналізується кожен з її компонентів.

2.1 Аналіз вимог до системи

Перед початком розробки будь-якого програмного рішення чи системи необхідно встановити початкові вимоги. Визначимо функціональні вимоги – набір задач або дій, які повинна виконувати система під час роботи:

- можливість конфігурування системи, в тому числі конфігурування допустимих значень лімітів та запитів при автоматичному режимі роботи;
- можливість збирати та аналізувати метрики утилізації ресурсів;
- можливість роботи в автоматизованому режимі;
- можливість відстежувати події аварійних завершень через нестачу ресурсів в кластері;
- можливість обчислення оптимальних лімітів для обчислювальних ресурсів;
- наявність GUI для отримання метрик, оптимальних значень для квот та їх застосування;
- загальні рекомендації, які стосуються оптимізації ресурсів.

Конфігурування системи має включити такі можливості, як задання цільових сервісів або розгортань для проведення моніторингу та визначення оптимальних лімітів для обчислювальних ресурсів. Також повинна бути можливість задання мінімальних та максимальних значень лімітів використання ресурсів при автоматизованому режимі роботи.

Збір та аналіз метрик утилізації ресурсів включає в себе автоматичне знаходження необхідних Prometheus метрик та подальшу агрегацію, аналіз та збереження у власному сховищі.

Автоматизований режим передбачає зменшення ролі адміністратора кластера Kubernetes при конфігуруванні лімітів обчислювальних ресурсів. Цей режим

передбачає не тільки аналіз та формування пропозиції по оптимальним лімітам, а також їх автоматичне встановлення згідно початкової конфігурації системи. У разі періодичного аварійного завершення додатку через нестачу ресурсів, система в автоматизованому режимі має підвищити значення квот ресурсів для коректної роботи сервісу.

Метрики використання ресурсів дозволяють отримати загальну картину по утилізації. Проте іноді трапляються аномальні періоди роботи, наприклад, різке збільшення запитів до сервісу, коли сервіс вимагає більше ресурсів, ніж задано в конфігурації. В таких випадках Kubernetes контролер починає завершувати роботу таких сервісів для того, щоб інші сервісі на фізичній машині могли функціонувати у нормальному режимі. Оскільки необроблені запити можуть залишитися в черзі, то після початку роботи нового екземпляру сервісу відбувається аналогічна ситуація – аварійне завершення. Тому необхідно реагувати на події завершення через нестачу ресурсів та прийняти міри – збільшити ліміти. Крім того, схожа ситуація з ресурсом процесора, але помилки при нестачі ресурсу не відбуваються. Замість аварійного завершення додаток починає штучно уповільнюватись з тою метою, аби використання процесора входило у вказаний ліміт та не заважало іншим додаткам на цьому поді.

Збір Prometheus метрик проводиться для того, щоб пізніше вирахувати оптимальні значення лімітів та сформувати пропозицію до адміністратора або відразу їх встановити. Обчислення оптимальних значень для RAM та CPU ресурсів необхідно проводити окремо за допомогою різних алгоритмамів через особливості конфігурування та використання в кластері Kubernetes.

Визначимо нефункціональні вимоги – властивості, які має демонструвати продукт, або обмеження, з якими необхідно працювати:

- система має працювати на будь-якому кластері Kubernetes незалежно від розміщення цього кластеру, кількості вузлів та з використанням лише загальнодоступних компонентів Kubernetes;
- система не повинна впливати на роботу кластера чи окремих сервісів, окрім конфігурування лімітів для обчислювальних ресурсів;

2.2 Сценарії використання системи

Далі розглядаютьс я можливі ситуації, коли дана система є корисною для використання у кластері:

- в існуючому кластері відсутні встановлені limit та request на CPU та пам’ять для сервісів. В такому випадку, Kubernetes застосує стандартні значення для всіх сервісів, встановлені при початковому конфігуруванні кластеру або простору імен [9]. Це призведе або до неоптимального використання ресурсів, або до аварійних завершень сервісів через нестачу ресурсів на фізичній машині. В даному випадку, система проаналізує використання ресурсів сервісами та встановить для них необхідні квоти на використання;
- в існуючому кластері не вистачає ресурсів для всіх сервісів. Необхідно оптимізувати квоти на ресурси;
- в існуючому кластері необхідно провести аналіз оптимальності встановлених квот на ресурси;
- в новому кластері, коли невідоме навантаження на сервіси та необхідно його оцінити.

Користувачами нашої системи можуть бути адміністратор кластеру та розробник, який бажає налаштувати моніторинг для свого мікросервісу. Далі розглядається які дії доступні кожному з них згідно з додатком В, на якому зображено діаграму прецедентів.

Серед доступних дій маємо наступні:

- перегляд подій, зокрема OOM;
- конфігурування системи, а саме інтервал запитів на сервер метрик, інтервал агрегування, режим роботи;
- додавання додатку до моніторингу, яке полягає в тому, щоб додати мікросервіс до списку цільових розгортань для моніторингу та керування, а також задати налаштування, такі як тип навантаження, максимальні та мінімальні значення для ресурсів;
- перегляд рекомендованих значень;

- встановлення рекомендованих значень, використовуючи графічний інтерфейс.

На даній діаграмі зображено дві дійові особи:

- адміністратор кластеру, який налаштовує дану систему в цілому;
- розробник, який відповідає за конкретний мікросервіс.

Так, наприклад, розробнику доступно лише додавання свого розгортання до списку цільових, але інтервал запитів до серверу метрик змінити не зможе через обмеження прав.

2.3 Розроблення структурної схеми системи

Структурна схема, що розробляється в даній магістерській дисертації, зображена на додатку Л. В цьому підрозділі розглядається кожен компонент цієї діаграми та обгрунтовується їх необхідність.

2.3.1 Блок збору метрик обчислювальних ресурсів Kubernetes

Даний сервер є джерелом метрик використання обчислювальних ресурсів для внутрішніх компонентів кластеру – планувальник, HPA, оцінювач ресурсів.

Для отримання даних використання для всіх вузлів кластеру необхідно виконати HTTP запит через kube-proxy на адресу apis/metrics.k8s.io/nodes, зображену на рисунку 2.2. Є можливість виконати аналогічний запит, але для отримання даних використання для всіх подів у кластері, або лише для одного простору імен.

```
GET /apis/metrics.k8s.io/v1beta1/nodes HTTP/1.1
Host: kubernetes-cluster:8001
Accept: application/json
```

Рисунок 2.2 – Запит на отримання метрик ресурсів вузлів кластеру

У відповіді отримаємо перелік всіх вузлів кластеру та їх поточні значення метрик використання ресурсів для кластеру, що зображено на рисунку 2.3:

```
"items": [
    {
        "metadata": {
            "name": "minikube",
            "creationTimestamp": "2020-11-22T19:04:34Z"
        },
        "timestamp": "2020-11-22T19:04:00Z",
        "usage": {
            "cpu": "557m",
            "memory": "964108Ki"
        }
    }
]
```


Рисунок 2.3 – Поточні значення метрик використання ресурсів для кластеру

В полі items отримуємо перелік вузлів, інформацію про них та метрики використання ресурсів. Поле metadata зберігає інформацію про вузол – його назву та дату створення, поле timestamp – час отримання метрик, usage – поточне використання метрик.

Далі наведено аналогічний запит, але для отримання метрик використання ресурсів для подів:

```
GET /apis/metrics.k8s.io/v1beta1/namespaces/default/pods HTTP/1.1
Host: kubernetes-cluster:8001
Accept: application/json
```

Рисунок 2.4 – Запит на отримання метрик використання ресурсів подами

Відповідь сервера аналогічна рисунку 2.4, формат даних еквівалентний, але замість інформації про вузли – інформація про подаи та контейнери.

2.3.2 Блок управління кластером

Даний сервер надає REST інтерфейс для взаємодії з кластером. За допомогою цього API можна створювати поди, розгортання та інші об’єкти Kubernetes, змінювати маніфести існуючих об’єктів, а також видаляти їх. Крім того, даний

інтерфейс дозволяє отримати стан всіх компонентів кластеру. Через даний сервер взаємодіють всі внутрішні компоненти кластеру.

Для системи, що розробляється, даний інтерфейс потрібен для отримання поточних квот на обчислювальні ресурси, а також для застосування нових значень. Крім того, даний компонент дозволяє відслідковувати події кластеру, зокрема, події аварійних завершень через нестачу пам’яті.

2.3.3 Клієнт Kubernetes

Клієнт дозволяє звертатись до Server API, використовуючи високорівневі операції, а не HTTP-запити. Також клієнт надає можливість отримувати події кластеру в реальному часі. Для розробки даної системи підходять kubectl та clientgo. Оскільки необхідно мати саме програмний інтерфейс, а kubectl є програмою з термінальним інтерфейсом, то залишається лише останній.

2.3.4 Блок збору та агрегування метрик

Сервер моніторингу збирає метрики за допомогою серверу метрик та зберігає їх в базі даних для подальшої обробки сервером рекомендацій. Також сервер моніторингу агрегує зібрані метрики за деякий час для оптимального використання бази даних в подальшому. Принцип роботи даного компоненту описано у наступних розділах.

2.3.5 Блок рекомендацій та автоматичного застосування квот

Сервер обчислень та рекомендацій на основі зібраних метрик обраховує рекомендовані значення квот для подів, а також автоматично їх застосовує. Крім того, даний сервер відстежує події аварійних завершень подів, для того, щоб підвищити квоти пам’яті при OOM помилці та уникнути таких відмов у подальшому. Принцип

роботи даного компоненту описано у наступних розділах. Алгоритм роботи даного блоку зображений на додатку Д.

2.3.6 Інтерфейс керування

Даний компонент відповідає за візуалізацію зібраних метрик, а також подій кластеру. Крім того, адміністратори кластеру можуть власноруч застосовувати квоти на ресурси за допомогою цього інтерфейсу, а також конфігурувати деякі аспекти роботи системи, наприклад, границі квот для встановлення або цільові простори імен в кластері.

2.3.7 База даних

База даних необхідна для зберігання метрик використання, квот на конкретний період, а також для зберігання подій. База даних може бути як реляційна, так і документо-орієнтовна. Вибір бази даних описано у наступному розділі.

2.4 Висновки

В даному розділі були описані вимоги до системи та сценарії використання. Структурна схема, розроблена в даному розділі, повністю відповідає поставленим вимогам. Також обґрунтовано необхідність кожного с компонентів даної структурної схеми.

3 АНАЛІЗ ТА ВИБІР ТЕХНІЧНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ

В даному розділі розглядаються технічні засоби реалізації даної системи, а саме:

- мова програмування – ключовий інструмент реалізації, від якого залежить архітектура системи, її продуктивність, здатність до масштабування, а також на набір бібліотек для роботи з Kubernetes;
- кластер для розробки – це середовище для вивчення та тестування властивостей системи, що розробляється;
- база даних – для зберігання метрик та їх подальшого аналізу необхідне надійне сховище для даних, а також для зберігання подій кластеру, в тому числі події OOM.

3.1 Вибір мови програмування

В даному розділі розглядаються сучасні мови програмування для реалізації системи. Різні мови програмування мають різні підходи, різні бібліотеки, здатність до масштабування та рівень продуктивності. Визначимо основні вимоги до мови програмування:

- підтримка асинхронних викликів, оскільки значна частина роботи сервера;
- це запити до серверу метрик, Kubernetes API та до бази даних. Використання асинхронного коду є оптимальним у порівнянні з синхронним кодом при роботі з інтенсивним вводом-виводом;
- набір бібліотек для роботи з Kubernetes – ключова вимога; без такої бібліотеки доведеться власноруч формувати запити на Server API, що є не складною, але рутинною роботою.

Серед мов, які розглядаються в даному розділі – Go, Python та Rust. Кожна з мов задовольняє наші вимоги та є актуальною на сьогодні. Також дані мови необхідно порівняти з точки зору швидкодії.

3.1.1 Аналіз мови програмування Python

Далі розглядається мова програмування Python. Python – інтерпретована, об’єктно-орієнтована мова програмування високого рівня. Дана мова використовує строгу динамічну типізацію. Наявність потужної вбудованої бібліотеки, велика спільнота та динамічна типізація роблять його дуже привабливим для швидкої розробки додатків, а також для написання скриптів. Лаконічний та простий у вивченні синтаксис Python робить код читабельним, а, отже, зменшує складність підтримки коду. Інтерпретатор Python та стандартна бібліотека доступні безкоштовно для всіх основних платформ і можуть вільно розповсюджуватися.

Оскільки в Python відсутній крок компіляції, цикл редагування-тестування-відлагодження є неймовірно швидким. Відлагодження програм Python є дуже простим, оскільки будь-які помилка ніколи не спричинить помилку сегментації. Доступний налагоджувач, який дозволяє перевіряти локальні та глобальні змінні, оцінювати довільні вирази, встановлювати точки зупинки, переходити через код за рядком за один раз тощо. Крім того, дана мова програмування є найпопулярнішою в 2020 році, що зображено на рисунку 3.1:

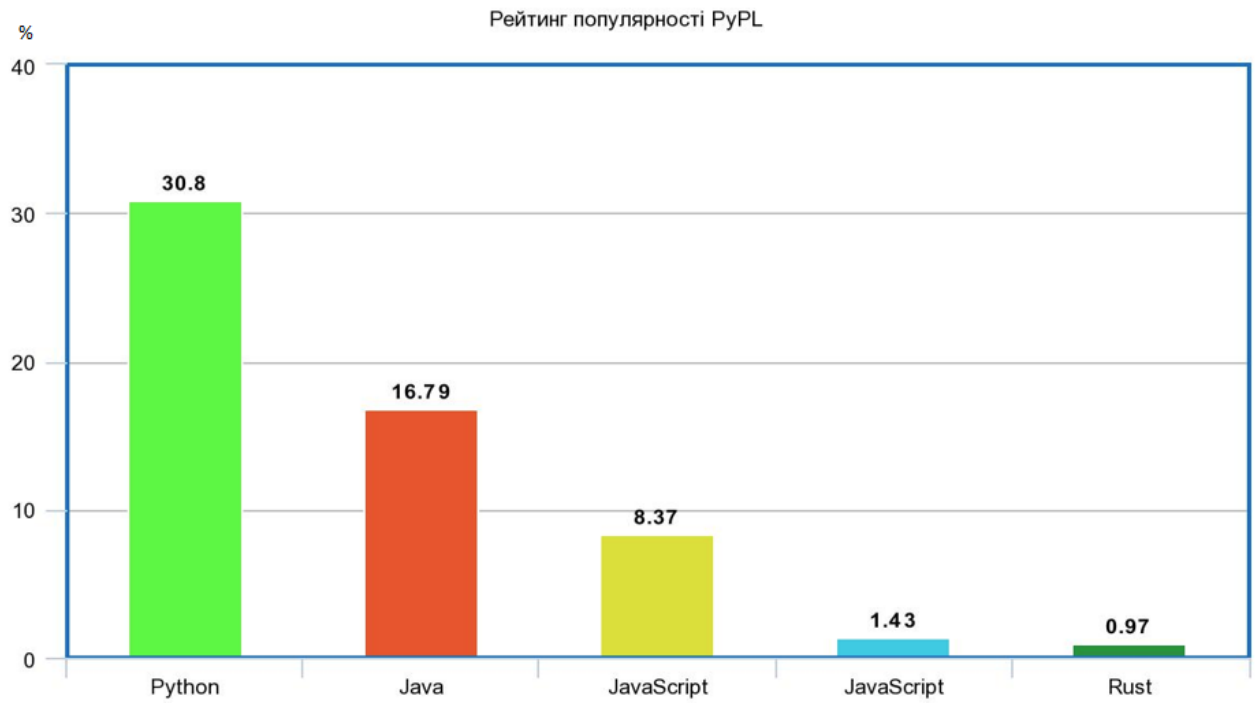

Рисунок 3.1 – Рейтинг популярності PyPL [31]

Основні переваги даної мови:

- проста у використанні, підходить для прототипування
- інтерпретатор, швидкий цикл редагування-тестування-відлагодження
- широкий вибір бібліотек
- асинхронна підтримка
- спільнота

Недоліки:

- повільна, оскільки інтерпретована
- не є безпечною в контексті типів
- проблеми при розробці мультипотокових програм

Дану мову необхідно розглянути в контексті розробки системи для моніторингу та керування обчислювальними ресурсами в кластері Kubernetes. Дана мова ідеально підходить для створення прототипу. Але в той же час є проблема з розробкою мультипотокових додатків, що у нашому випадку є важливим. Крім того, бібліотека для роботи з Kubernetes не така потужна, як в мові програмування Go.

3.1.2 Аналіз мови програмування Go

Далі розглядається мову програмування Go. Go – це статично-типізована, компільована мова програмування, розроблена в Google. Go синтаксично схожий на C, але також включає безпечну роботу з пам'яттю, збиранням сміття та вбудованою конкурентністю.

Дана мова програмування проста у використанні та читанні. Можливо, Go не є такою популярною, як JavaScript або Python, але дана мова програмування входить у 20 найрозповсюдженіших за рейтингом PyPL, і одна із головних причин – Go проста у використанні та розумінні.

Синтаксис Go досить простий, а полога крива навчання робить його доступним для програмістів початківців. Наприклад, Go має необхідний мінімум вбудованих функцій для використання, які мають прості сигнатури. Дана мова унаслідувала риси від C, C# та C++, а це значить, що вивчення Go для більшості програмістів має бути

легким.

Потужна стандартна бібліотека. Розробники на Go мають доступ до потужної стандартної бібліотеки, яка доступна від початку користування, що допомагає уніфікувати код та зменшити необхідність у використанні сторонніх бібліотек.

Безпека. Простий код є безпечнішим, оскільки в ньому складніше зробити помилку, що призведе, наприклад, до segfault. Більш того, оскільки мова має статичну строгу типізацію, розробникам на Go не доводиться турбуватися про помилки через неправильний формат даних або тип – проблеми, які часто виникають у мов з динамічною типізацією.

Відсутність проблем з пам’яттю – збирач сміття постійно перевіряє об’єкти на актуальність та видаляє не потрібні для запобіганню проблем з пам’яттю. Широка та детальна документація – хоча основна перевага Go полягає в простоті написання та читання, але документація завжди є важливою частиною будь-якої мови програмування. Дана мова добре документована – як стандартні вбудовані типи та функції, так і бібліотеки.

Переваги даної мови:

- простота використання;
- швидкість – дана мова є компільованою, що свідчить про високу швидкість виконання;
- вбудована підтримка конкурентності, що дає можливість використовувати асинхронний підхід;
- безпека на рівна роботи з пам’яттю та типами;
- добре підходить для написання системних програм, наприклад, Kubernetes та Docker написані саме на Go.

Недоліки:

- складна та нестандартизована обробка помилок;
- відсутність generic-типів, що призводить до дублювання коду та порушення принципу DRY.

Необхідно розглянути дану мову в контексті розробки системи для моніторингу та керування обчислювальними ресурсами в кластері Kubernetes. Дана мова найкращу

інтеграцію з Kubernetes, оскільки сам Kubernetes та Docker написані на Go, тому розробники підтримують клієнтські біблітки для даної мови в першу чергу. Можливість застосування асинхронного підходу, простота в розробці та читанні, інтеграція з Kubernetes робить дану мову найбільш доцільною для розробки додатків для Kubernetes.

3.1.3 Аналіз мови програмування Rust

Також розглядається мова програмування Rust – мова, яка розрахована на розробку саме низькорівневих, швидких, системних додатків. Rust – це статична і сильнотипізована мова програмування. Основною особливістю даної мови є відсутність «garbage collector», що значно пришвидшує виконання програм.

Основні переваги даної мови:

- продуктивність, що зображено на рисунку 3.2;
- система типів;
- підтримка асинхронного програмування;
- безпека пам’яті.

Серед недоліків:

- складний синтаксис;
- не підходить для прототипування;
- відсутність інтеграції з Kubernetes.

3.1.4 Порівняння мов у контексті використання ресурсів

Порівняємо дані мови по швидкої та використанню пам’яті. На рисунку 3.2 зображено графік обчислення хешів для всіх трьох мов, де можна побачити, що Python дуже повільний у порівнянні з двома іншими, а Go та Rust знаходяться приблизно на одному рівні.

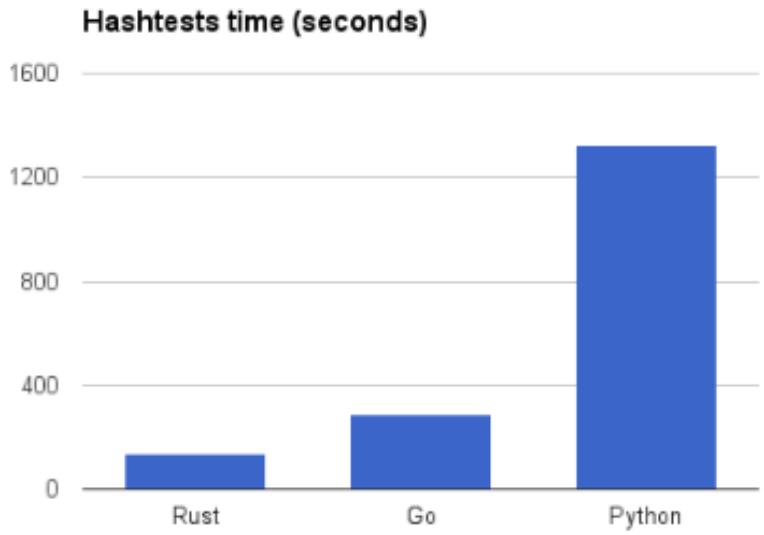

Рисунок 3.2 – Швидкість обчислення функції хешування на різних мова

Також порівняємо використання пам’яті даними мовами програмування. На рисунку 3.3 показано дане порівняння. Бачимо, що Python потребує досить багато пам’яті, а Go і Rust всього приблизно 1 Мб.

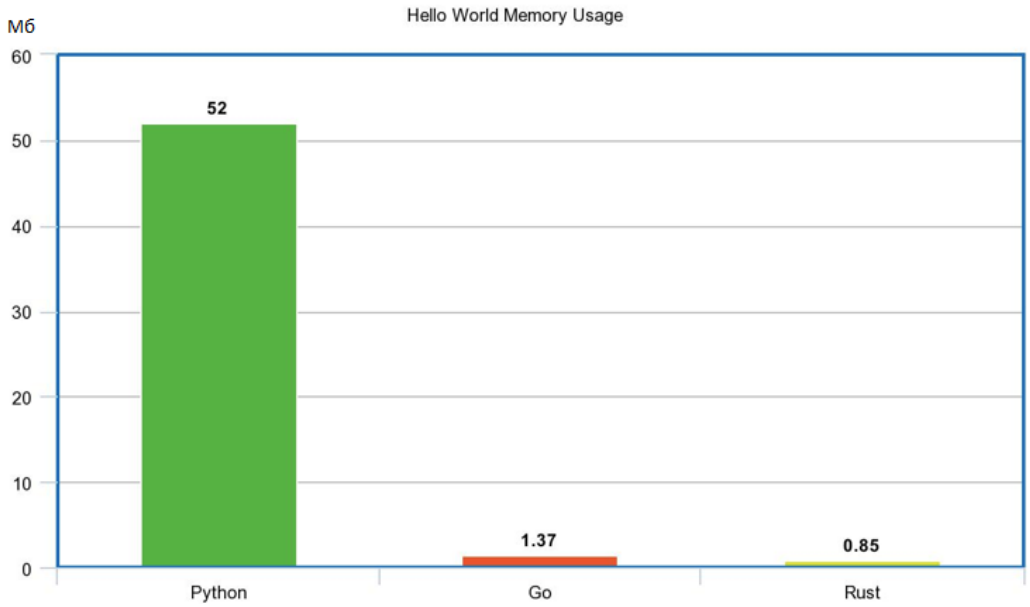

Рисунок 3.3 – Використання пам’яті базовою програмою

3.2 Вибір кластеру Kubernetes для розробки та тестування

Середовище для розробки та тестування дає змогу оцінити правильність роботи системи. Визначимо вимоги до середовища:

- це має бути повноцінний кластер Kubernetes зі всіма активними компонентами;
- повний доступ до цього кластеру, оскільки в процесі розробки в даний кластер необхідно встановити розширення та постійно розгортати додатки для тестування;
- вигідно з точки зору фінансів;
- кількість вузлів не є важливою, оскільки кластер може працювати всього з одним вузлом.
- На основі сформованих вимог обираємо серед двух рішень – кластер Kubernetes на основі Google Cloud Platform та Minikube, який працює на локальній машині.

3.2.1 Аналіз Google Kubernetes Engine

Перший варіант – на основі Google Cloud Platform. Google Kubernetes Engine (GKE) забезпечує середовище для розгортання, управління та масштабування додатків на основі контейнерів за допомогою використання інфраструктури Google [32]. Середовище GKE складається з декількох машин (зокрема, екземплярів Compute Engine), об’єднаних в кластер за допомогою приватної мережі. Варто зазначити, що GKE автоматично генерує готовий до роботи кластер та налаштовує вузли, а також бере на себе відповідальність, якщо компоненти вийдуть з ладу.

Кластери GKE працюють на основі загальнодоступної версії Kubernetes. Взаємодія з кластером відбувається стандартними методами для роботи з будь-яким Kubernetes кластером – утиліти командного рядка kubectl. Крім стандартних компонентів Kubernetes, GKE автоматично налаштовує такі компоненти:

- балансування навантаження за допомогою Google Cloud для вузлів на базі Compute Engine;
- пули вузлів для позначення підмножин вузлів у кластері для додаткової гнучкості;
- автоматичне масштабування кількості екземплярів вузла кластера;

- автоматичне оновлення програмного забезпечення вузлів кластера;
- автоматичне відновлення вузла при збоях в роботі або недоступності вузла.

Далі розглядається приклад налаштування кластеру. Для того, щоб створити кластер в GKE необхідно в панель розробника вибрати пункт GKE, після чого кластер можна створити всього одним кліком:

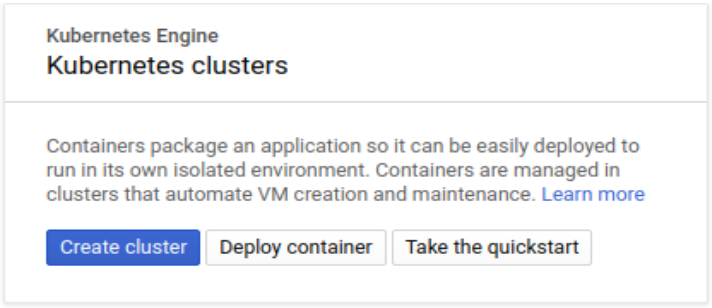

Рисунок 3.4 – Створення кластеру GKE

Далі необхідно його налаштувати, серед налаштувань, що зображено на рисунку 3.5:

- ім’я кластеру;
- тип розташування, де можна вибрати фізичне місце перебування серверів, а також можна вказати декілька регіонів;
- версія Kubernetes.

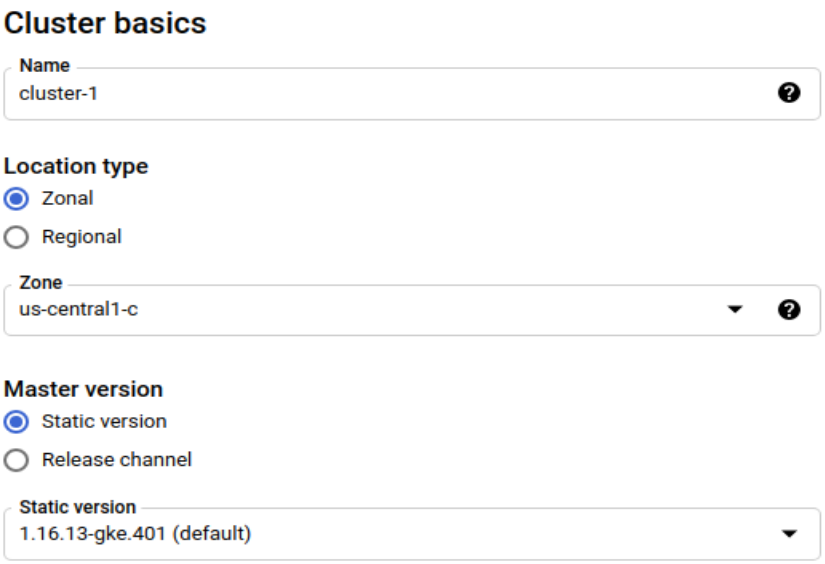

Рисунок 3.5 – Налаштування кластеру GKE

Також в кластері можна налаштувати Istio, що зображено на рисунку 3.6, що є складним завданням навіть для досвідченого адміністратора. Крім того, є можливість налаштувати TPU – tensor processor unit для задач навчання нейронних мереж.

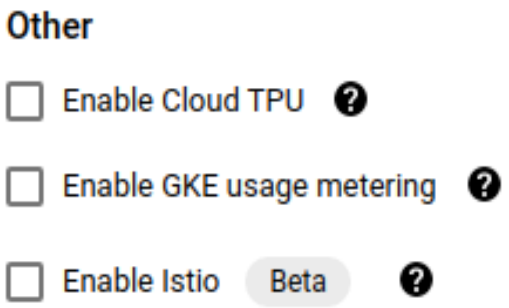

Рисунок 3.6 – Додаткові налаштування

В нашому випадку Istio та TPU не є актуальними, але така можливість є унікальною серед хмарних варіантів Kubernetes.

3.2.2 Аналіз Minikube

Наступний інструмент, який розглядається – це Minikube. Minikube – це локальний кластер Kubernetes на macOS, Linux та Windows. Основними цілями Minikube є полегшити розробку додатків для Kubernetes. Minikube є повноцінним Kubernetes кластером, який включає всі необхідні компоненти, такі як сервер метрик та kube-proxy, що будуть необхідними при розробці [33].

Єдиним вузлом даного кластеру є локальна машина, тому варто пам'ятати про обмеження розміщення кількості подів на одному вузлі. На цьому вузлі знаходяться як системні контролери Kubernetes, так і користувацькі додатки, тому локальна машина має бути досить потужною, для того, щоб кластер працював у нормальному режимі.

Детальніше його налаштування описано в наступному розділі, оскільки в даному підрозділі обирається саме дане рішення як більш оптимальне з точку зору зручності, економічних витрат та швидкості роботи.

3.3 Вибір бази дани

Для зберігання метрик та обчислених рекомендацій в системі необхідно мати базу даних. Сучасні бази даних можна розділити на два типи – реляційні та нереляційні, а також на підтипи – реляційні, документо-орієнтовані, ключ-значення, графові та інші. Для створення прототипу необхідна база даних, з якою просто працювати, швидко розгортається та в подальшому її можна масштабувати для реальних навантажень.

3.3.1 Аналіз бази даних PostgreSQL

PostgreSQL – це система система управління реляційними базазами даних. PostgreSQL підтримує як запити SQL (реляційні), так і JSON (нереляційні). Дана база даних має підтримку мов програмування за допомогою розширень, функцій, тригерів, індексування даних. Вона є досить швидкою та має можливість масштабування за схемою master-slave. Крім того, дана база даних має переваги перед аналогами, серед яких MySQL, SQLite:

- визначені користувачем типи;
- наслідування таблиць;
- механізм lock-синхронізацій;
- підтримка views, правил, підзапитів;
- підтримка вкладених транзакцій;
- асинхронна реплікація.

Дана база відповідає всім вимогам нашої системи – швидкість роботи, масштабованість, наявність транзакція та можливість індексувати дані.

3.3.2 Аналіз бази даних Cassandra

Apache Cassandra – це розподілена NoSQL база даних, яка використовує розподілену модель зберігання широких стовпців, на відміну від PostgreSQL, яка

використовує модель зберігання через строки. Дана модель дозволяє значно швидше обробляти запити агрегації даних, що є актуальною перевагою в процесі роботи систем моніторингу.

Серед переваг даної бази даних:

- масштабованість, яка, на відміну від PostgreSQL, розповсюджується на запис, а не тільки на читання;
- швидкий запис та швидке читання;
- швидка обробка агрегацій за рахунок використання моделі стовбців.

Недоліки:

- складність у розгортанні;
- неповна підтримка транзакцій;
- затримка між вузлами кластеру Cassandra.

Оскільки це розподілена бази даних, то в ній не реалізовано механізм транзакцій у повному обсязі. Крім того, Cassandra працює в повну силу з декількома вузлами, а PostgreSQL достатньо master-вузла. Також ця база даних складна в обслуговуванні, тому вважаю, що перший варіант є оптимальнішим.

3.4 Висновки

В даному розділі були проаналізовані технічні засоби реалізації, а саме мова програмування, а також середовище для локальної розробки та тестування.

В розділі 3.1 були розглянуті такі мови програмування як Python, Go и Rust. На основі аналізу обрано саме мову програмування Go, оскільки дана мова потужну бібліотеку для роботи з Kubernetes, асинхронну модель роботи на базі горутин, а також простий синтаксис, що надає змогу швидко написати прототип системи. Крім того, Kubernetes та Docker написані саме на цій мова програмування. Також перевагами даної мови є швидкість роботи та безпека типів і пам’яті.

В розділі 3.2 розглянуто середовища для розгортання Kubernetes кластеру для розробки та тестування. Розглядалося два рішення – Google Kubernetes Engine та Minikube. Серед основних переваг останнього:

- кластер знаходиться на локальній машині, а це означає, шо можна використати локальний Docker реєстр образів, що значно полегшує розгортання власних додатків у кластері;
- дане рішення включає всі необхідні компоненти, в тому числі сервер метрик, що є ключовим компонентом для реалізації системи моніторингу та керування обчислювальними ресурсами;
- безкоштовно, на відміну від GKE, який потребує фінансових витрат для додавання вузлів до кластеру.

PostgreSQL обрано як основну базу даних, оскільки вона значно простіша в роботі та обслуговуванні, підтримує транзакції та може масштабуватися при потребі.

Отже, для розробки обрано мову програмування Go, а як середовище для розробки та тестування – кластер Kubernetes на базі Minikube.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

В даному розділі описується розроблення програми, алгоритму для знаходження оптимального значень квот, а також налаштування середовища для розробки та тестування.

4.1 Розгортання кластеру для розробки

У розділі 3.2 розглянуто рішення для розгортання кластеру та обрано Minikube, як досить компактну і безкоштовну систему. Далі розглядається як розгортається Minikube. Вимоги до машини, на якій розгортається Minikube [33]:

- 2 процесорних ядра або більше;
- 2 Гб оперативної пам’яті;
- 20 Гб вільного місця на диску;
- підключення до інтернету;
- диспетчер контейнерів або віртуальних машин, таких як: Docker, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox або VMWare.

Фізична машина, на якій піднімається кластер має 4 ядра, 16 Гб оперативної пам’яті, достатньо вільного місця на диску, а також KVM для віртуалізація.

Встановлюємо кластер за допомогою команди sudo dpkg –i minikube_latest_amd64.deb, після чого запускаємо кластер командою minikube start. Результат зображений на рисунку 4.1, де видно, що піднімається Kubernetes, а також його компоненти – вузол, необхідні для роботи контейнери та деякі розширення – dashboard, default-storageclass, metrics-server. Компонент dashboard дозволяє розгортати поди, переглядати стан кластеру, контролювати події кластеру, а також встановлювати квоти для розгортань. Компонент default-storageclass необхідний для додатків, які використовують файлову систему. Метричний сервер є вимкненим на початку роботи кластеру, тому його необхідно додати до списку бажаних розширень, що розглянуто далі.

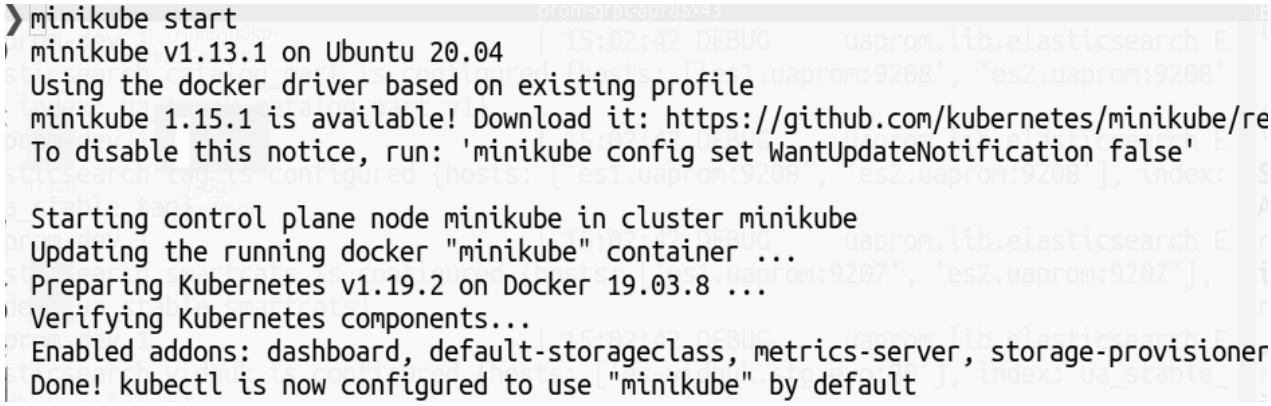

Рисунок 4.1 – Запуск кластеру Minikube

Для розробки необхідне розширення серверу метрик, яке на початку є вимкненим, що зображено на рисунку 4.2. Вимкнено майже всі додатки, оскільки Minikube розрахований для запуску на одній машині, тому ресурси необхідно економити.

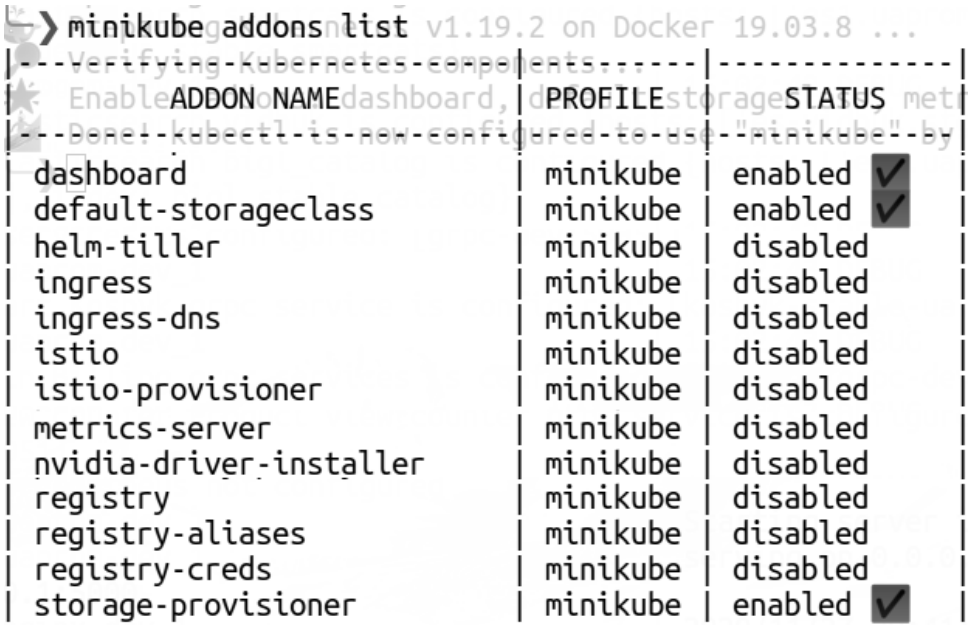

Рисунок 4.2 – Список розширень кластеру Minikube

Метричний сервер вмикається командою minikube addons enable metrics-server, що зображено на рисунку 4.3. Після виконання бачимо повідомлення про успішне завершення, а отже метричний сервер встановлено.

```
└❯ minikube addons enable metrics-server
    The 'metrics-server' addon is enabled
```

Рисунок 4.3 – Додавання серверу метрик до кластеру Minikube

Для того, щоб отримати системні компоненти даного кластеру необхідно виконати команду kubectl get pod,svc –n kube-system, яка виводить всі поди в системному просторі імен. На рисунку 4.4 бачимо такі компоненти:

- etcd-minikube – сховище даних кластеру;
- kube-scheduler-minikube – планувальник Kubernetes, який розглядався у розділі 1;
- metrics-server-d9b576748-99dzr – сервер метрик, який використовується у даній роботі для отримання метрик;
- kube-apiserver-minikube – сервер API Kubernetes, який також використовується для отримання поточних квот використання ресурсів.

```
└❯ kubectl get pod,svc -n kube-system
NAME                                   READY   STATUS    RESTARTS
pod/coredns-f9fd979d6-nnt79            1/1     Running   4
pod/etcd-minikube                      1/1     Running   3
pod/kube-apiserver-minikube            1/1     Running   3
pod/kube-controller-manager-minikube   1/1     Running   5
pod/kube-proxy-9th7r                   1/1     Running   4
pod/kube-scheduler-minikube            1/1     Running   4
pod/metrics-server-d9b576748-99dzr     1/1     Running   0
pod/storage-provisioner                1/1     Running   16
```

Рисунок 4.4 – Компоненти кластеру

Отже, отримано повністю працюючий кластер з ввімкненим сервером метрик, що зображено на рисунку 4.4. Цей кластер містить всього один вузол, що зображено на рисунку 4.5:

```
└❯ kubectl get nodes
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   21d   v1.19.2
```

Рисунок 4.5 – Вузли кластеру

4.2 Особливості роботи планувальника Kubernetes

Для кожного щойно створеного пода або інших ресурсів планувальник Kubernetes має обрати оптимальний вузол для розміщення. Кожен контейнер у пода має різні вимоги до ресурсів, і кожен под також має різні вимоги. Тому всі доступних вузли потрібно відфільтрувати відповідно до конкретних вимог планування. Вузли кластеру, що відповідають вимогам планування для поду, називаються потенційними вузлами розміщення. Якщо жоден з вузлів не підходить по вимогам, то под залишається в статусі «назапланований» до тих пір, поки планувальник не зможе його розмістити. Планувальник знаходить підходящі під вимоги вузли для поду, а потім запускає процес оцінки, на основі якого обраний єдиний вузол. Потім планувальник повідомляє Server API про це рішення. Даний алгоритм зображений на додатку Е.

4.3 Алгоритм розміщення подів

Фактори, які потрібно враховувати при планування, включають індивідуальні та сумарні вимоги до ресурсів, обмеження на апаратне, програмне забезпечення та політику розміщення, специфікації спорідненості та антифінності (anti-affinity), прив’язку до даних, тощо [25].

Планувальник обирає вузол в 2 етапи:

1. Фільтрування
2. Підрахунок балів

На першому кроці планувальник перевіряє, чи має вузол-кандидат достатньо доступних ресурсів для задоволення запитів ресурсів поду. Якщо після цього кроку список вузлів містить хоча б один вузол, то планувальник переходить до наступного етапу, інакше под чекатиме вільний вузол для розміщення.

Фактично планувальник викликає команду kubectl describe node для кожного активного вузла і таким чином фільтрує потрібні. На рисунку 4.6 зображено приклад виконання команди.

```
Capacity:
  cpu:                          8
  ephemeral-storage:            229700940Ki
  evo.company/network-slots:    1
  hugepages-1Gi:                0
  hugepages-2Mi:                0
  memory:                       16245148Ki
  pods:                         110
Allocatable:
  cpu:                          7
  ephemeral-storage:            211692385954
  evo.company/network-slots:    1
  hugepages-1Gi:                0
  hugepages-2Mi:                0
  memory:                       15618460Ki
  pods:                         110
```

Рисунок 4.6 – Результат виконання команди kubectl describe node

На рисунку 4.6 бачимо характеристику вузла, а саме місткість – capacity, яка говорить про ресурси вузла в цілому, а доступні для розміщення – allocatable – ті, які може використати планувальник при розміщенні подів. Доступні ресурси є меншими за місткість, оскільки частина ресурсів виділена під хостову операційну систему.

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource                    Requests       Limits
  --------                    --------       ------
  cpu                         6860m (98%)    19 (271%)
  memory                      13254Mi (86%)  17006Mi (111%)
  ephemeral-storage           0 (0%)         0 (0%)
  hugepages-1Gi               0 (0%)         0 (0%)
  hugepages-2Mi               0 (0%)         0 (0%)
  evo.company/network-slots   1              1
```

Рисунок 4.7 – Використання ресурсів вузла

На рисунку 4.7 бачимо скільки поточних ресурсів використовується, а саме процесорного ресурсу – 98%, а пам’яті – 86%. Тоді при розміщенні фільтруванні вузлів віднімаємо від доступних для використання (allocatable) поточне використання: 7000m – 6860m = 140m. Тобто на цей вузол можна ще розмістити один мікросервіс с допустимим використанням процесора не більше 140m. Под зі специфікацією на рисунку 4.8 на даному вузлі розміститися зможе:

```
resources:
      limits:
        cpu: 100m
        memory: 170Mi
      requests:
        cpu: 100m
        memory: 170Mi
```

Рисунок 4.8 – Часткова специфікація поду

На етапі підрахунку, планувальник ранжує вузли, щоб вибрати найбільш підходящий. Призначається оцінку кожному вузлу, який пройшов фільтрування. Потім планувальник розміщує под на вузлі з найвищим рейтингом. Якщо є більше одного вузла з однаковими оцінками, вибирається один із них випадковим чином.

Kubernetes дозволяє обмежити для поду потенційні вузли для розміщення, або віддавати перевагу розміщенню на певних вузлах. Це можна зробити декількома способами за допомогою використання спеціальних міток – nodeSelector. Як правило, такі обмеження є зайвими, оскільки планувальник автоматично розміщує на вузлах с потрібною кількістю ресурсів, але існують деякі обставини, коли може знадобитися більше контролю, наприклад, щоб переконатись, що под потрапить на машину з твердотільним накопичувачем, або до групи близько розміщених вузлів, між якими є інтенсивна комунікація, в одній і тій же зоні доступності.

```
affinity:
   nodeAffinity:
       nodeSelectorTerms:
       - matchExpressions:
         - key: feature
           operator: In
           values:
           - ssd
           - raid10
```

Рисунок 4.9 – Приклад специфікації с nodeSelector

Мітка nodeSelector – це найпростіша та рекомендована форма обмеження вибору вузла. Це поле PodSpec, яке визначає право поду розміщуватися на вузлі.

Якщо при плануванні поду немає доступного вузла з такою міткою, то под не буде розміщений. Також на цей вузол можуть розміститись і поди, без такої мітки, але лише у випадку недостатності ресурсів на вузлах без міток.

Також вище згадувалось про фінність/антифінність – це інструмент для того, щоб рекомендувати планувальнику, але не забороняти. Так, наприклад, можна рекомендувати планувальнику розміщувати мікросервіс на машинах, де присутній RAID із жорстких дисків, як зображено на рисунку 4.9, але якщо такого вільного вузла немає, планувальник розмістить под на без RAID.

Спрощено схема вибору вузла виглядає так:

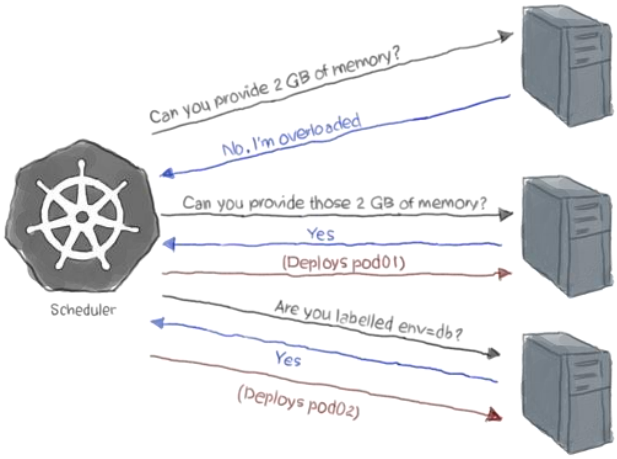

Рисунок 4.10 – Схема вибору вузла планувальником [26]

Планувальник, як зображено на рисунку 4.10, знаходить вузол з достатніми ресурсами та необхідними мітками, а потім робить оцінку на базі фінності\афінності та інших факторів.

4.4 Отримання метрик кластеру

Сервер метрик знаходиться в кластері, для розробки є можливість роботи запити на нього за меж кластеру використовуючи команду kubectl proxy, яка використовує компонент kube-proxy, що міститься в переліку компонентів на рисунку 4.4. Схема отримання метрик зображена на додатку Ж. Даний проксі сервер працює

на адресі localhost:8001, та дозволяє роботи запити до будь-яких компонентів в середині кластеру. Отримаємо поточне споживання ресурсів по вузлам в кластері використовуючи запит на рисунку 4.11:

```
1  GET /apis/metrics.k8s.io/v1beta1/nodes HTTP/1.1
2  Host: localhost:8001
3  Accept: application/json
```

Рисунок 4.11 – Запит на використання ресурсів по вузлам

Результат на рисунку 4.12, де можна побачити поточне споживання єдиного вузла – 475 мілікорів процесорного часу та 1.2 Гб оперативної пам’яті.


```
{
    "metadata": {
        "name": "minikube",
        "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/minikube",
        "creationTimestamp": "2020-11-28T13:28:03Z"
    },
    "timestamp": "2020-11-28T13:27:00Z",
    "window": "1m0s",
    "usage": {
        "cpu": "475m",
        "memory": "1174932Ki"
    }
}
```


Рисунок 4.12 – Поточне використання ресурсів вузлами кластеру

Тепер отримаємо аналогічні дані, але для використання ресурсів контейнерами. На рисунку 4.13 зображено частковий результат запиту.


```
{
    "name": "kube-scheduler",
    "usage": {
        "cpu": "2m",
        "memory": "24100Ki"
    }
}
```


Рисунок 4.13 – Використання ресурсів планувальником Kubernetes

Бачимо, що компонент kube-scheduler використовує всього 2 мілікори процесора та 24 Мб оперативної пам’яті. Саме таким чином будуть отримуватися метрики для всіх мікросервісів, а в подальшому аналізуватися.

4.5 Алгоритм розрахунку квот на ресурси

Навантаження на ресурси можна розділити на декілька сценації, що трапляються найчастіше:

- постійно зростаюче використання одного з ресурсів;
- періодичне зростання та зменшення одного з ресурсів;
- відсутність будь-якого патерну поведінки;

Постійно зростаючий ресурс є типовим патерном та трапляється в будь-якій інформаційній системі. Далі наводиться приклад. База даних Redis, яка зберігає всі дані в оперативній пам’яті для швидкої роботи з ними.

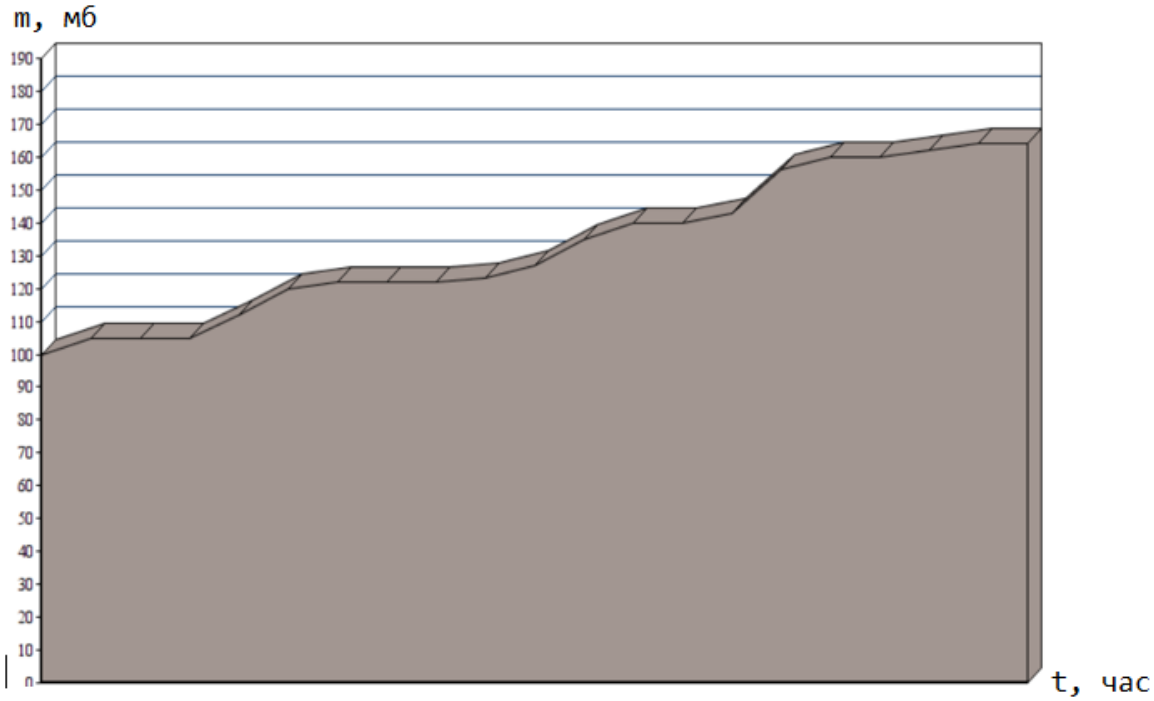

Рисунок 4.14 – Приклад постійно зростаючого використання пам’яті

Дуже часто дана база даних використовується для зберігання користувацький сесій, а якщо ваша система розвивається, то і кількість сесій в базі теж зростатиме,

що призведе до постійно зростаючого використання оперативної пам’яті. В такому випадку, квоти на ресурси необхідно постійно контролювати та слідкувати за OOM помилками. Приклад графіку використання зображений на рисунку 4.14. Для тких випадків виділимо спеціальний клас для такого типу сервісів – high-availability або високої доступності. Для такого випадку система керування та моніторингу в автоматичному режимі ніколи не буде знижувати квоти на ресурс, але тільки підвищувати в випадку потреби або OOM помилки завершення. Якщо використання ресурсу все ж таки знизиться, то в такому випадку адміністратор кластеру отримує сповіщення, що ресурс даного сервісу необхідно зменшити вручну – через графічну панель системи. Ситуація, коли регулювання пам’яті за допомогою VPA призводило до аварійних не є рідкістю, тому виділяємо саме такий клас навантажень.

Обробка даного сценарію дозволить автоматизувати керування ресурсами мікросервісу з навантаженням такого типу, що не є доступним у таких аналогів як VPA та KubeOptimizer.

Наступний сценарій – це періодичне підвищення та зменшення споживання ресурсів, що зображено на рисунку 4.15. Такий сценарій також не є рідкістю в сучасних інформаційних системах.

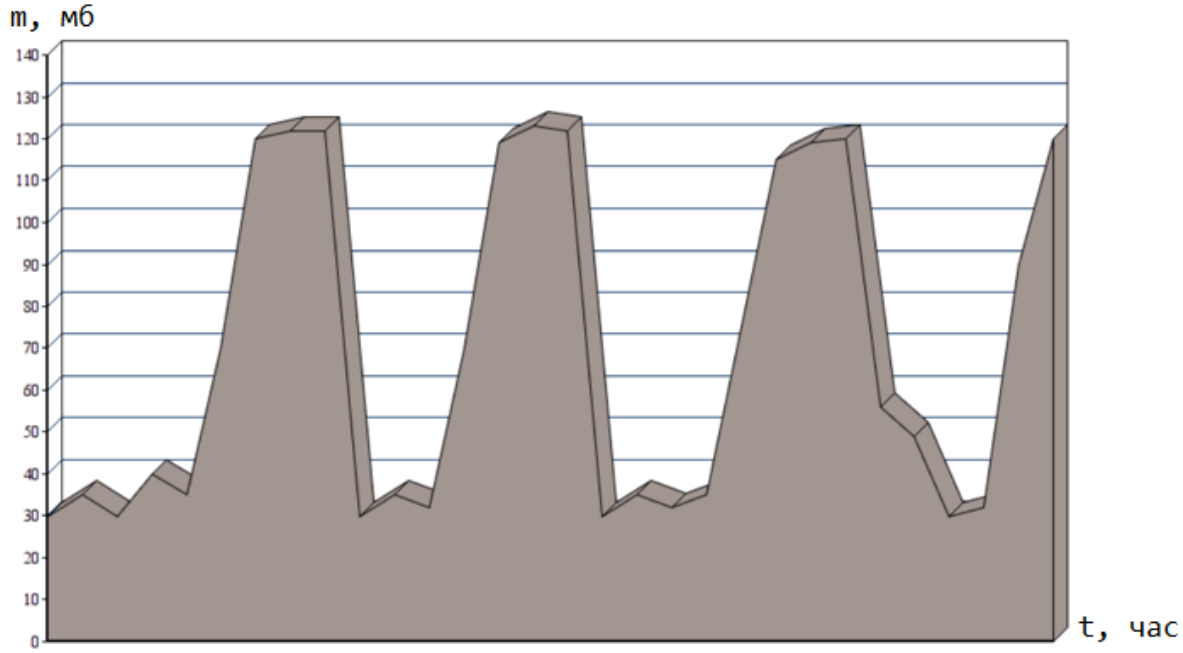

Рисунок 4.15 – Періодичне навантаження

Такий сценарій виникає при використанні асинхронних задач для обробки даних, які запускаються з періодом. Наприклад, кожні дві години робиться клієнтська email-розсилка. І сервер для надсилання email повідомлень отримує навантаження лише кожні дві години. Або якщо деяка розсилка робиться раз в 3 дні, тоді не має сенсу тримати обчислювальні ресурси кластеру на мікросервісі, який простоює. У такому випадку, на основі аналізу метрик використання необхідно з'ясувати даний період навантаження, та відразу виділяти ресурси саме в ці періоди. Це дозволить значно зменшити середнє використання ресурсів цим сервісом, а отже дозволить уникнути фінансовим витрат під час простоювання.

Останній та найскладніший тип навантаження, коли відсутні будь-які патерни. Це найчастіший випадок, притаманний веб-застосункам. Наприклад, користувачів на порталі то більше, то менше і кількість не залежить від внутрішніх факторів. Даний тип навантаження зображений на рисунку 4.16:

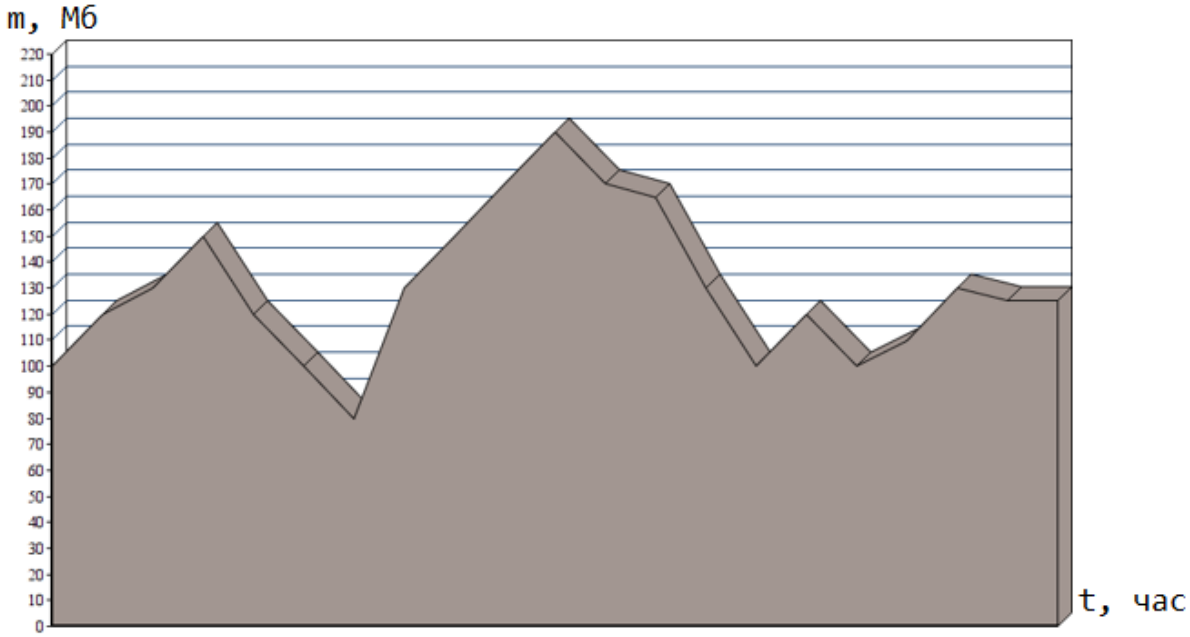

Рисунок 4.16 – Навантаження, яке не залежить від внутрішніх факторів

Далі розглядається алгоритм дій в такому випадку. При потребі або помилках OOM підвищити квоти на ресурси відразу. Якщо навантаження почало зменшуватися, то необхідно впевнитись, що пік навантаження пройшов, для цього можемо почекати, наприклад, 30 хвилин.

Далі розглядається яким чином вираховуються конкретні значення для квот. Нехай отримано поточні значення використання для обох ресурсів – процесора та пам’яті. Для ресурсу процесора необхідно обрати таке значення, аби його використання 95% часу було на межі ліміту. Значення 95% вибрано емпіричним шляхом, оскільки при нестачі ресурсу процесора додаток не припиняється, а лише штучно сповільнюється, то можна допустити таку ситуацію, що використання процесору на короткий час перевищує допустимий ліміт. Тоді ті 5% потраплять до наступного періоду розрахунку процесорного часу і таким чином можна заощадити на ресурсі процесора. При цьому це заощадження становитиме значно більше ніж 5%. Наприклад, на рисунку 4.16 ліміт можна встановити за максимальним значенням графіка – 195, тоді весь інший час процесор простоюватиме, адже в середньому йому потрібно 160 мілікорів. А якщо ліміт встановити в 180, тоді в піковому навантаженні додаток трохи сповільниться, але отримує свій час в наступному часовому періоді. При усереднені, отримуємо 20 збиткових мілікорів, а при максимальному – 35 мілікорів, тоді можливо заощадити 40% невикористаних ресурсів. Ці 60% – це компроміс між продуктивністю в пікових навантаженнях та економії ресурсів на довгій дистанції.

Якщо говорити про оптимальні квоти для ресурсу пам’яті, то необхідно пам’ятати, що при перевищенні додатком ліміту, він завершиться з OOM помилкою. Тому необхідно значення ліміту брати по максимальному використанню пам’яті за деякий період. Також додаємо невеликий запас, для того щоб випадкова алокація пам’яті не призвела до OOM помилок. Такий запас є значенням, що можна налаштувати, та рекомендується встановлювати приблизно в 10% від максимального споживання.

4.6 Розроблення структури програми

В даному розділі розглядається структура програми. На додатку М зображена діаграма розгортання застосунку, що розробляється. Серед основних компонентів

діаграми – Kubernetes, застосунок та база – PostgreSQL Server. Kubernetes має два модулі – metrics-server та API.

Сервер метрик, або metrics-server – це загальнокластерний агрегатор даних використання ресурсів. Він збирає такі показники, як споживання процесора або пам'яті для контейнерів або вузлів через API Summary, що надаються через Kubelet на кожному вузлі. Minikube має даний сервіс за замовчуванням. При звертанні через HTTP запит до спеціальної адреси apis/metrics/v1beta1/namespaces/default/pods отримуємо поточні значення використання ресурсів у JSON вигляді, що описано у розділі 1.

Модуль API – це загальний інтерфейс Kubernetes для отримання даних про стан кластеру та зміни специфікацій. Саме через цей інтерфейс працює kubectl. Для роботи з API Kubernetes розробники надають готову до використання бібліотеку go-client.

Даного функціоналу достатньо для того, щоб знаходити сервіси та розгортання, а також встановлювати для них квоти на ресурси. Більш того, за допомогою цієї бібліотеки можливо підписатися на всі події в кластері, що необхідно для реагування на аварійні події через нестачу ресурсів.

Компонент DataScrapper звертається до metrics-server для отримання метрик, після чого обробляє та зберігає їх в базі даних. Період опитування сервера метрик є конфігурабельним та має співпадати з періодом оновлення метрик самим сервером метрик. Значення поточного використання зберігаються окремо для кожного контейнеру кожного поду, а потім агрегується для розгортань. Крім того, даний підписаний на всі події в кластері і зберігає, якщо отримано подію OOM, що свідчить про недостачу пам’яті для поду.

На рисунку 4.17 показано спрощений варіант коду роботи компоненту DataScrapper. Необхідно зазначити, що частина коду зв’язана з обробкою помилок та контейнерів в даному прикладі була видалена. Алгоритм роботи виглядає так:

1. Отримуємо поточне використання з серверу метрик.
2. Якщо отримали помилку під час запиту, то чекаємо час для повторення та повертаємося до кроку 1.
3. Кожне отримане значення записуємо до бази даних.

4. Чекаємо деякий час, потім повертаємося до кроку 1.

В даному коді retryInterval – це час, який даний компонент чекатиме для того, щоб зробити повторний запит в разі помилки. А workerScrapInterval – це інтервал оновлення значень на сервері метрик. В даному коді опрацьовані можливі помилки при роботі з базою даних та сервером метрик. Значення retryInterval та workerScrapInterval є частиною конфігурації та можуть задаватися адміністратором кластеру.

```
func RunWorker(db *sqlx.DB) {
    for {
        err, resp := scrap()
        if err != nil {
            time.Sleep(retryInterval)
            continue
        }
        for _, item := range resp.Items {
            container := item.Containers[0]
            database.InsertMetricRecord(db, container)
        }
        time.Sleep(workerScrapInterval)
    }
}
```

Рисунок 4.17 – Код роботи компоненту DataScrapper

Компонент Recommender-Updater відповідає за аналіз метрик та подій, зібраних через DataScrapper. Для аналізу на предмет оптимальних значень квот, даному компоненту також необхідно отримати дані з API Kubernetes, а саме поточні значення лімітів та запитів. Даний компонент отримує поточний стан по квотам через API Kubernetes та аналізує зібрані метрики, після чого за одним із алгоритмів рекомендує нові значення, а також встановлює їх в автоматичному режимі. Алгоритм роботи вже був описаний в розділі 4.5.

Компонент Dashboard – це панель адміністратора, де можна побачити графіки по зібраним метрикам, а також переглянути поточні значення та рекомендації. Для роботи використовується API Kubernetes, для того, щоб отримати поточні квоти, а також база даних для читання метрик та рекомендацій.

На рисунку 4.18 зображена спрощена діаграма компонентів програми. Компоненти Recommender, DataScrapper та Dashboard вже розглядалися, тому тут аналізується лише їх взаємодію з іншими компонентами.

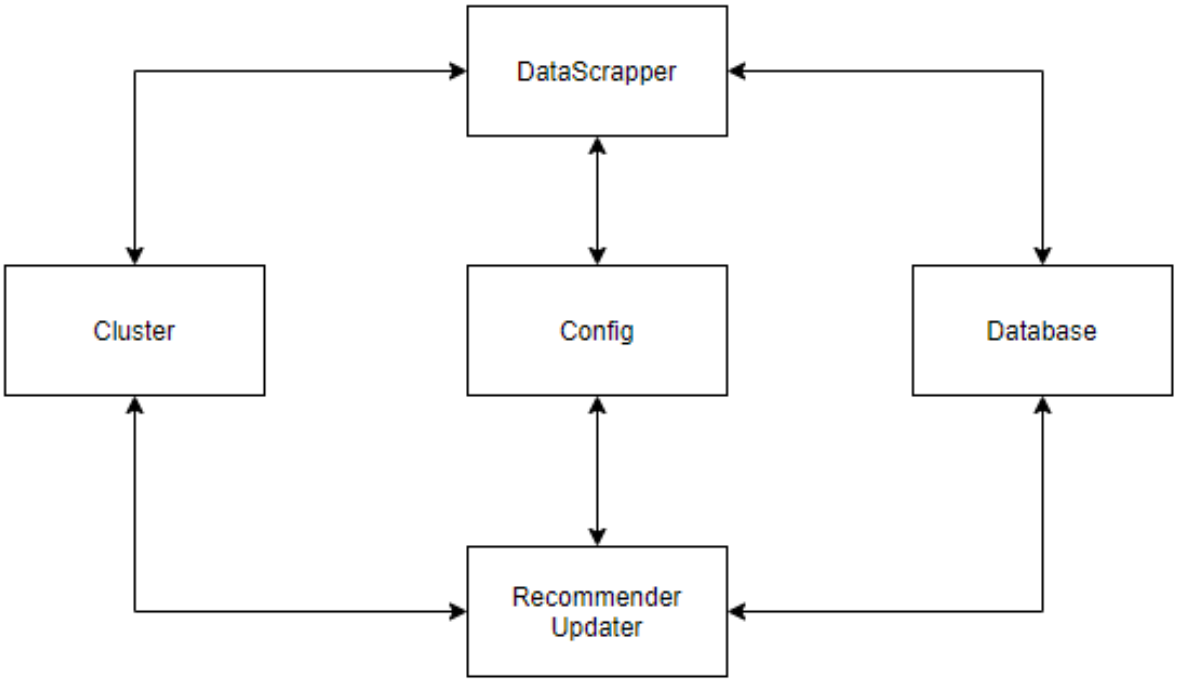

Рисунок 4.18 – Спрощена діаграма компонентів

Компонент Database відповідає за роботи з базою, а саме з’єднання та виконання запитів. Для роботи с PostgreSQL використовується бібліотека sqlx, яка надає всі необхідні функції. Також компонент містить містить всі моделі, а саме:

- MetricRecord для зберігання та роботи з метриками;
- Deployment для роботи з розгортаннями;
- Recommendation для запису обрахованих значень для квот;
- Events для роботи с подіями в кластері.

Даний компонент використовується в Recommender, DataScrapper та Dashboard, оскільки кожен з них потребує читати або записувати в базу даних. Так, наприклад, Dashboard використовує для читання агрегованих метрик, а також для читання рекомендованих значень. Компонент DataScrapper записує всі отримані метрики, а компонент Recommender обраховує значення для квот використовуючи дані з таблиць та зберігає обраховані дані.

Компонент Dashboard містить графічний інтерфейс, який дозволяє переглядати результати роботи системи.

Компонент Cluster містить бібліотеку методів для отримання даних з кластеру, а також модифікації розгортань для встановлення оптимальних значень. Наприклад, на рисунку 4.19 зображено метод для отримання поточних значень квот. Аргумент name – це назва розгортання, а значення яке повертається містить два поля, а саме ліміти та запити для процесора та пам’яті.

```
func GetDeploymentResourceQuota(name string) corev1.ResourceRequirements {
    deploymentsClient := Client.AppsV1().Deployments(v1.NamespaceDefault)
    deployment, err := deploymentsClient.Get(context.TODO(), name, v1.GetOptions{})
    if err != nil : err.Error() *
    return deployment.Spec.Template.Spec.Containers[0].Resources
}
```

Рисунок 4.19 – Метод для отримання поточних значень квот

Компонент Config містить бібліотеку для роботи з конфігурацією системи та складається з конфігурації DataScrapper та конфігурації Recommender.

4.7 Розроблення структури бази даних

Далі розглядається структура бази даних. Діаграму сутностей можна побачити на додатку Г. Система містить такі таблиці:

- MetricRecord для зберігання значень використання контейнерами ресурсів, структура якої зображено в таблиці 4.1;
- Recommendation для зберігання обчислених значень квот для розгортань;
- Event містить події кластеру, а саме OOM, які необхідні для корегування обчислених лімітів та швидкої реакції на аварійні завершення мікросервісів через нестачу ресурсів;
- Deployment для роботи з розгортаннями;
- MetricRecordAgg містить агреговані значення використання ресурсів контейнерами за деякий час.

Таблиця 4.1 – Опис MetricRecord

Назва поля	Тип	Призначення
id	integer	Унікальний ідентифікатор запису
deployment_id	integer	Ідентифікатор розгортання
pod_name	text	Ідентифікатор поду
container_name	text	Назва контейнера всередині поду
cpu	integer	Використання процесора в мілікорах
memory	integer	Використання оперативної пам’яті в кілобайтах
timestamp	integer	Час, коли дана метрика була отримана

Оскільки таблиці MetricRecord може містити досить багато записів, оскільки зберігає значення використання ресурсів для кожного контейнеру як мінімум кожні 60 секунд, то робота з даною таблицею буде повільна, та скоріш за все частина застарілих даних буде видалятися через завеликий об’єм даних. Тому використовується таблиця MetricRecordAgg, яка містить агреговані значення метрик за деякий час, наприклад, за годину або за день. Ця таблиця значно пришвидшує роботу системи та спрощує відображення графіків використання ресурсів. Дана таблиця майже повністю аналогічна до MetricRecord, але містить два додаткові поля timestamp_start та timestamp_end. Структура відображена в таблиці 4.2.

Таблиця 4.2 – Опис MetricRecordAgg

Назва поля	Тип	Призначення
id	integer	Унікальний ідентифікатор запису
deployment_id	integer	Ідентифікатор розгортання, до якого даний запис належить

Назва поля	Тип	Призначення
pod_name	text	Ідентифікатор поду
event_type	text	Тип події
cpu	integer	Використання процесора в мілікорах
memory	integer	Використання оперативної пам’яті в кілобайтах
timestamp_start	integer	Початок періоду агрегування
timestamp_end	integer	Кінець періоду агрегування

Сутність Event відображає події кластеру, необхідні для корегування рекомендацій. Її структуру відображена в таблиці 4.3. Наприклад, OOM помилка свідчить про те, що мікросервісу не вистачає оперативної пам’яті та ліміт потрібно збільшити.

Таблиця 4.3 – Опис Event

Назва поля	Тип	Призначення
id	integer	Унікальний ідентифікатор події
deployment_id	integer	Ідентифікатор розгортання, до якого дана подія належить
pod_name	text	Ідентифікатор поду
container_name	text	Назва контейнера всередині поду
timestamp	integer	Час, коли подія сталася

В таблиці Recommendation містяться рекомендовані значення, отримані в результаті аналізу споживання ресурсів. В таблиці 4.4 бачимо, що зв’язок з сутністю

Deployment один до одного, це говорить, що в рекомендаціях знаходяться лише актуальні значення.

Таблиця 4.4 – Опис Recommendation

Назва поля	Тип	Призначення
id	integer	Унікальний ідентифікатор рекомендації
deployment_id	integer	Ідентифікатор розгортання Deployment
Назва поля	Тип	Призначення
pod_name	text	Ідентифікатор поду
container_name	text	Назва контейнера всередині поду
timestamp	integer	Час, коли подія сталася
cpu	integer	Рекомендоване значення ліміту для процесора в мілікорах
memory	integer	Рекомендоване значення ліміту для пам’яті в кілобайтах

Сутність Deployment містить всього два поля, що зображено в таблиці 4.5, та є як зв’язуюча між всіма іншими. Наявність запису в цій таблиці свідчить, що для розгортання потрібно проводити моніторинг. Дана таблиця у подальшому буде містити налаштування для обчислення рекомендованих значень, наприклад, мінімально можливі квоти, або максимальні межі.

Таблиця 4.5 – Опис Deployment

Назва поля	Тип	Призначення
id	integer	Унікальний ідентифікатор розгортання
name	text	Назва розгортання

4.8 Розгортання системи

Далі розглядається як розгортається дана система для тестування. Діаграма розгортання системи зображена в додатку М. Для роботи системи в тестовому режимі необхідно:

- доступ до бази даних;
- доступ до кластеру Kubernetes через kube-proxy;
- розгорнути тестові додатки в кластері.

Для підняття бази даних використовується docker-compose, який дозволяє швидко запускати додатки використовуючи docker. Далі розглядається конфігурацію docker-compose на рисунку 4.20. Для того щоб запустити тестову базу даних достатньо виконати команду docker-compose up db.

```
services:
  pg:
    image: "postgres"
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: metrics
    volumes:
      - dbdata:/var/lib/postgresql/data/

volumes:
  dbdata:
```

Рисунок 4.20 – Конфігурація docker-compose для бази даних

Для того щоб у додатку був доступ до кластеру, необхідно запустити проксуючий сервер командою kubectl proxy, яка на 8001 порту відкриває доступ до кластеру.

Наступним кроком є запуск тестових розгортань. Для цього додалися розгортання, в яких поди обчислюють постійно число Фібоначчі з різною величиною та періодом, що зображено на рисунку 4.21.

```
└❯ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
fibgo-65fcdcf7b9-gc7k8              1/1     Running   3          20d
fibgo-65fcdcf7b9-lnk5p              1/1     Running   3          20d
fibgo-65fcdcf7b9-vt4wc              1/1     Running   3          20d
fibgo-ez-6d8796858b-chwl4           1/1     Running   4          21d
fibgo-ez-6d8796858b-fgsxg           1/1     Running   4          21d
```

Рисунок 4.21 – Тестові розгортання обчислень

Для запуску системи її необхідно скомпілювати, а потім виконати бінарний файл, що зображено на рисунку 4.22. Бачимо, що піднімається графічний інтерфейс на адресі localhost:8888, а також бачимо запити до серверу метрик і результати по тестовим подам.

```
~/projects/k8s-metrics-scrapper master !1 ····
└❯ go build -o metrics cmd/*.go

~/projects/k8s-metrics-scrapper master !1 ?1 ····
└❯ ./metrics
2020/11/29 01:24:22 Listening on 8888
2020/11/29 01:24:22 Worker is inited {url=http://localhost:8001/apis/metrics.k8s.io/v1beta1/namespaces/default/pods, interval=5s}
2020/11/29 01:24:22 Inserting metrics: fibgo-ez-6d8796858b-fgsxg, fibgo-ez, 57, 8444
2020/11/29 01:24:22 Inserting metrics: fibgo-65fcdcf7b9-lnk5p, fibgo, 59, 8668
2020/11/29 01:24:22 Inserting metrics: prometheus-1606165789-pushgateway-76c77959d8-qh4rg, prometheus-pushgateway, 0, 10152
2020/11/29 01:24:22 Inserting metrics: fibgo-65fcdcf7b9-vt4wc, fibgo, 62, 7636
2020/11/29 01:24:22 Inserting metrics: fibgo-ez-6d8796858b-chwl4, fibgo-ez, 65, 8316
2020/11/29 01:24:22 Inserting metrics: fibgo-65fcdcf7b9-gc7k8, fibgo, 57, 7928
```

Рисунок 4.22 – Компіляція та виконання програми

Можемо по адресі localhost:8888 отримати візуалізовані зібрані метрики по тестовим розгортанням, що і зображено на рисунку 4.23.

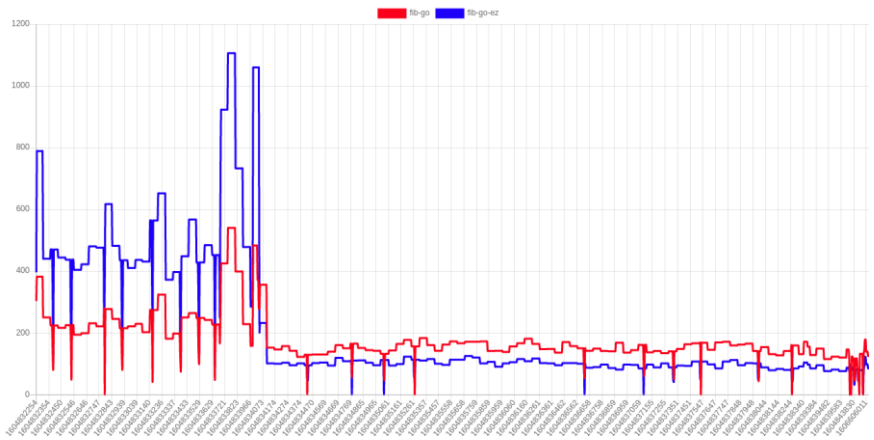

Рисунок 4.23 – Візуалізація зібраних метрик в графічного інтерфейсі

Далі наведений приклад застосування розрахованих значень квот для тестового розгортання fibgo:

```
2020/11/29 02:11:49 Recommended values for deployment fibgo {Cpu=152m, Mem=27421Ki}
2020/11/29 02:11:49 Updating limits and requests for deployment fibgo {Cpu=152m, Mem=27421Ki}
2020/11/29 02:11:49 Current value for fibgo Mem = 27421Ki Cpu = 152m
```

Рисунок 4.24 – Застосування обчислених квот на ресурси

Переглянемо стан подів для даного розгортання, щоб упевнитися, що ліміти та запити дійсно застосовувалися за допомогою команди kubectl describe pods:

```
Limits:
  cpu:     152m
  memory:  27421Ki
Requests:
  cpu:        152m
  memory:     27421Ki
```

Рисунок 4.25 – Застосовані квоти на ресурси

Перевіримо, чи перезапустив планувальник Kubernetes поди після зміни квот на ресурси:

```
❯ kubectl get pods
NAME                      READY   STATUS        RESTARTS   AGE
fibgo-6977969448-mbzvj    1/1     Running       0          9s
fibgo-6977969448-ndxrf    1/1     Running       0          3s
fibgo-6977969448-xrph2    1/1     Running       0          6s
fibgo-7d78bfd967-2bjdc    0/1     Terminating   0          13m
fibgo-7d78bfd967-gxhrl    1/1     Terminating   0          13m
```

Рисунок 4.26 – Автоматичний перезапуск подів після змін квот на ресурси

4.9 Висновки

В даному розділі розглянуто програмну реалізацію автоматизованої системи моніторингу та керування обчислювальними ресурсами в кластері Kubernetes, визначено вимоги до системи, а також сценарії використання.

В розділі 4.3 описано алгоритм розгортання кластеру Kubernetes для розробки та тестування за допомогою Minikube, а також встановлення необхідних для розробки розширень, а саме серверу метрик та kube-proxy. Розглядались основні компоненти Minikube та їх застосування.

В розділі 4.4 розглянуто алгоритм роботи з сервером метрики, описані запити для отримання поточних метрик для вузлів кластеру, а також подів, наведені приклади роботи з даним сервером в кластері, розгорнутому в розділі 4.3, а також наведені приклади отриманих метрик.

В розділі 4.5 розглянуто алгоритми для обчислення рекомендованих значень для лімітів на використання ресурсів додатками, а також типи навантажень. Крім того, описано особливості роботи з ресурсами процесору та пам’яті при встановленні лімітів.

Також розглянуто архітектуру системи, компоненти та взаємодію між ними, наведено структурну діаграму та діаграму розгортання, розглянуто схему бази даних та описано призначення таблиць та їх полів.

В розділі 4.8 описано розгортання розробленої системи, а також проведено тестування застосування обчислених квот.

5 ТЕСТУВАННЯ СИСТЕМИ

В даному розділі проводиться тестування розробленої системи моніторингу керування обчислювальними ресурсами в кластері Kubernetes. Також проводиться порівняння з одним з існуючих рішень, а саме Vertical Pod Autoscaler разом з Goldilocks. Схема тестовго середовища зображена в додатку К.

В даному розділі розглядаються наступні функції системи:

- моніторинг всіх існуючих мікросервісів у кластері;
- зменшення квот на ресурси при збитковому резервуванні;
- збільшення квот на ресурси при їх недостачі.

5.1 Тестування функції моніторингу

Розроблена система при старті автоматично знаходить всі користувацькі розгортання в заданому просторі імен та збирає для них метрики використання, що зображено на рисунку 5.1, де можна побачити перелік всіх розгортань:

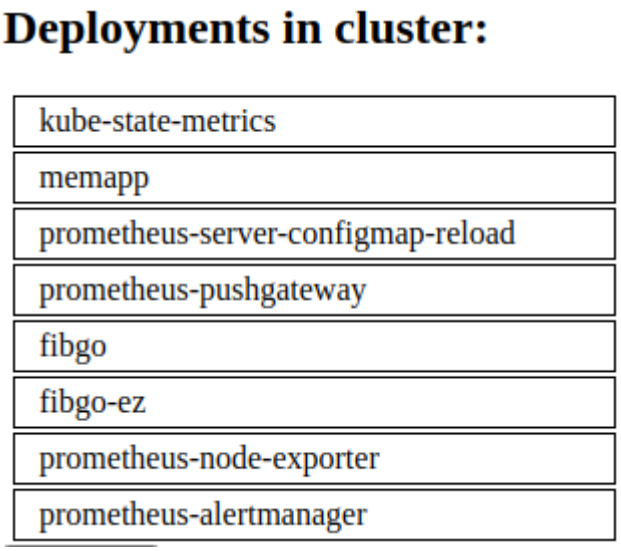

Рисунок 5.1 – Автоматично знаходження всіх розгортань кластеру

На рисунку 5.1 можна також побачити розгортання системи моніторингу Prometheus та kube-state-metrics. Це сталося, оскільки розроблена системи аналізує всі

простори імен, окрім системного – kube-system. При розгортанні Prometheus був вказаний простір імен – default, саме тому він потрапив до цієї виборки.

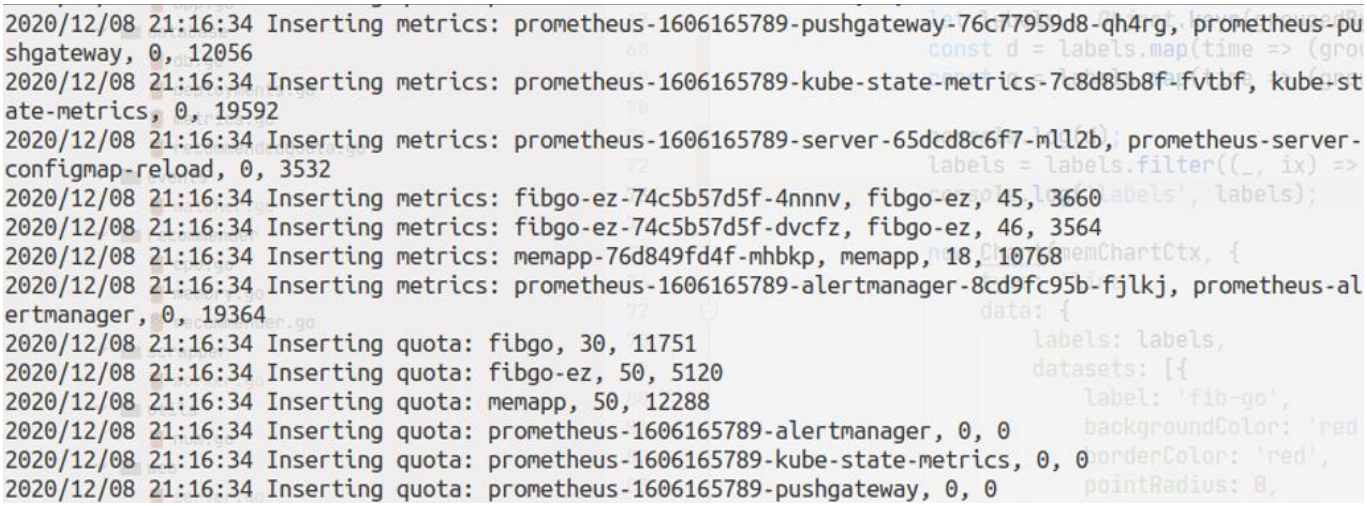

Рисунок 5.2 – Журнал збору метрик

На рисунку 5.2 бачимо журнал збору метрик, також поточних квот для того, щоб візуалізувати їх на графіках. Метрики збираються для кожного поду, а потім агрегуються через розгортання. На рисунку 5.2 перша назва – це назва поду, наступною є використання процесора в мілікорах, останньою є використання пам’яті в кілобайтах. Період збору метрик становить 15 секунд.

На рисунку 5.3 бачимо візуалізовані зібрані метрики використання ресурсів та квоти на них, а саме використання та квоти пам’яті тестового мікросервісу fibgo з рисунку 5.1. Вісь абсцис відповідає за відображення значення використання в конкретний момент часу в кілобайтах. Початкове значення на цій вісі є найменше значення використання за обраний період. Вісь ординат відображає час, за який отримано метрики та квоти. На рисунку видно, що період отримання метрик становить 15 секунд, але є можливість його налаштувати. Інтервал краще встановлювати відповідно до інтервалу опитування серверу метрик.

Червоний графік відображає залежність використання пам’яті мікросервісом від часу. Синій графік відповідає за значення квот на момент отримання метрик – саме для цього система і збирає значення лімітів та запитів. На даному рисунку це просто пряма, оскільки ліміти за час моніторингу не змінювалися.

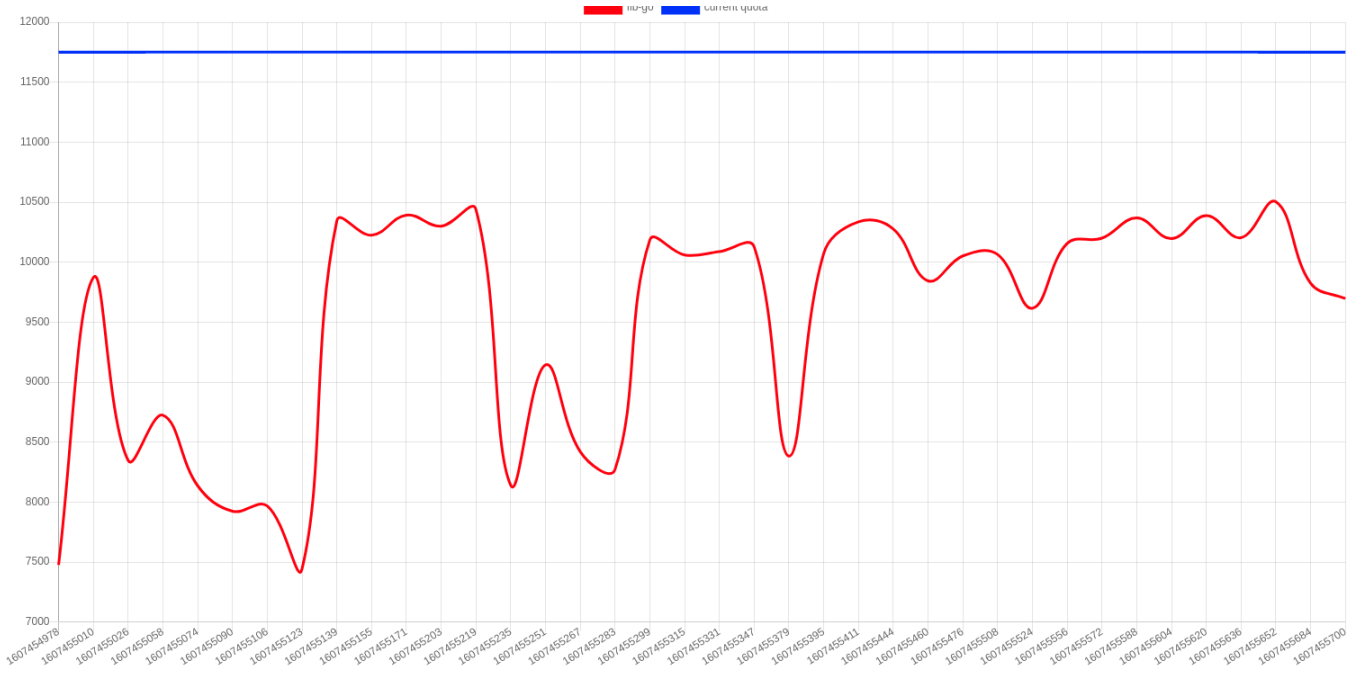

Рисунок 5.3 – Візуалізація зібраних метрик

Графіки, зображені на рисунку 5.3 можуть бути автоматично отримані для будь-якого розгортання за будь-який час, але в межах роботи системи, в кластері. Для цього не потребується додаткова конфігурація. У разі відсутності встановлених квот для розгортання, синій графік буде відсутній.

5.2 Тестування функції автоматичного керування ресурсами

На рисунках 5.4 та 5.5 зображено приклад автоматичного управління ресурсами – для процесора та пам’яті.

На першому рисунку зображений графік використання процесора. Використання знаходяться в межах від 70 до 130, але 90% часу не перевищує 105 мілікорів. Синій графік відображає поточну квоту, а саме 150 мілікорів. Зрозуміло, що квоти є збитковими і необхідно їх зменшити. Як бачимо, система автоматично призначила нову квоту в 105 мілікорів. Виникає питання, що відбудеться під час піку в 230 мілікорів використання. В такому випадку додаток штучно сповільнюється,

оскільки такий пік виник лише раз за час моніторингу, то він суттєво не впливає на рекомендоване значення.

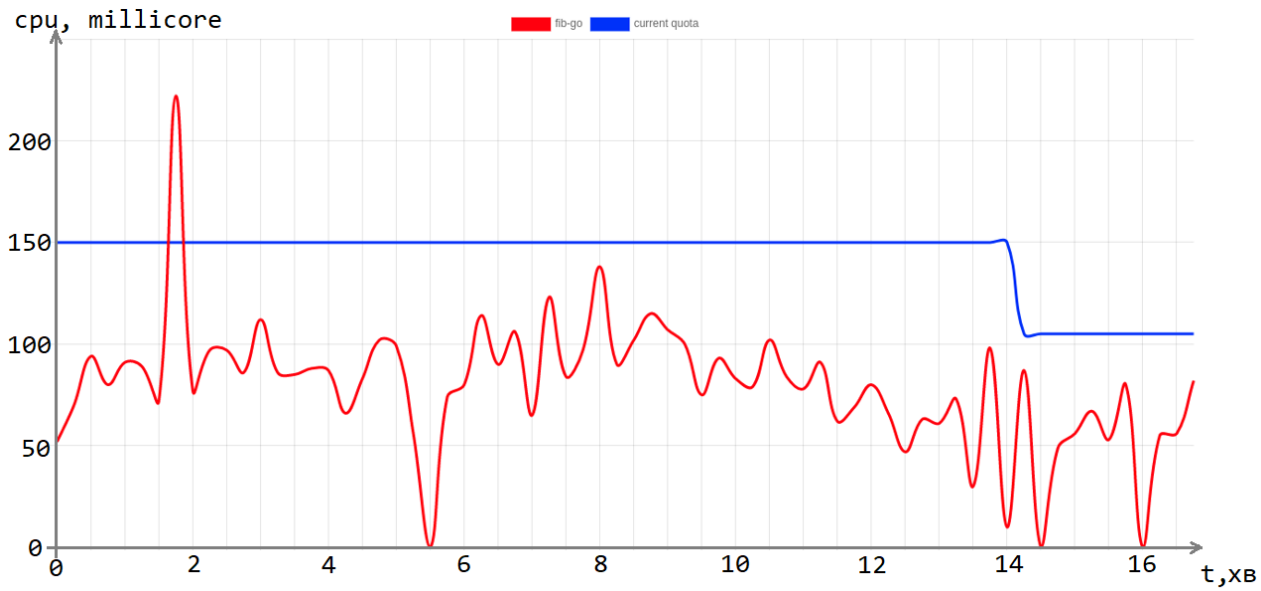

Рисунок 5.4 – Оптимізація запиту процесора

На рисунку 5.5 зображено процес оптимізації квот для пам’яті. Бачимо, що споживання додатку становить на рівні 8 Мб. Тому оптимальним значенням є приблизно 8 Мб та деякий запас. Отже, квота встановлена в 10 Мб.

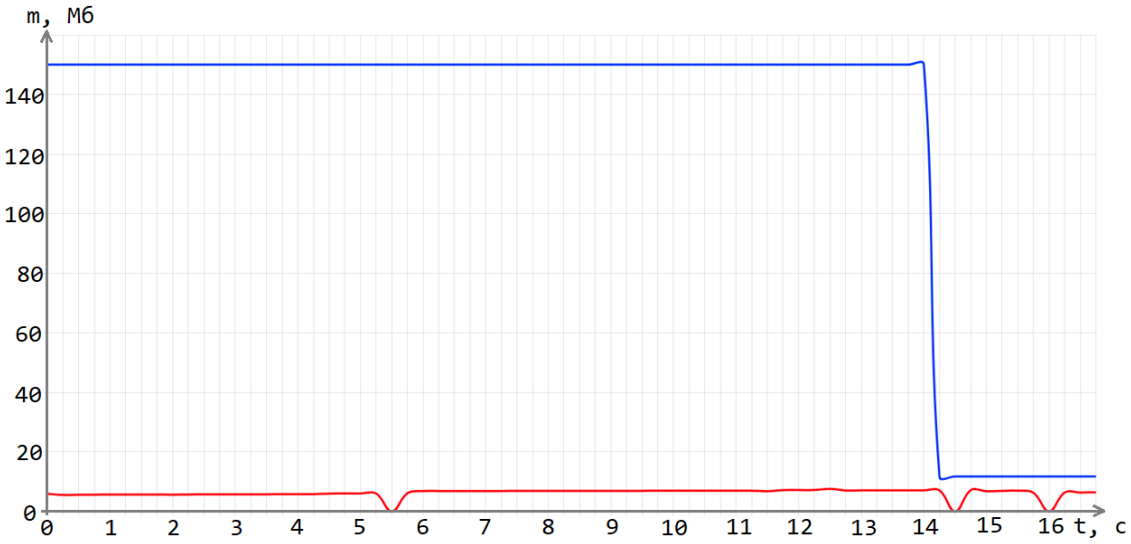

Рисунок 5.5 – Оптимізація запиту пам’яті

На рисунку 5.7 зображено використання процесора в діапазоні від 28 до 31 мілікорів до 5 хвилини. Можна зробити висновок, що квоту необхідно підвищити, оскільки додаток штучно сповільняється. На 6 хвилині відбувається підвищення квоти на використання процесора, що дозволяє додатку працювати більш стабільно.

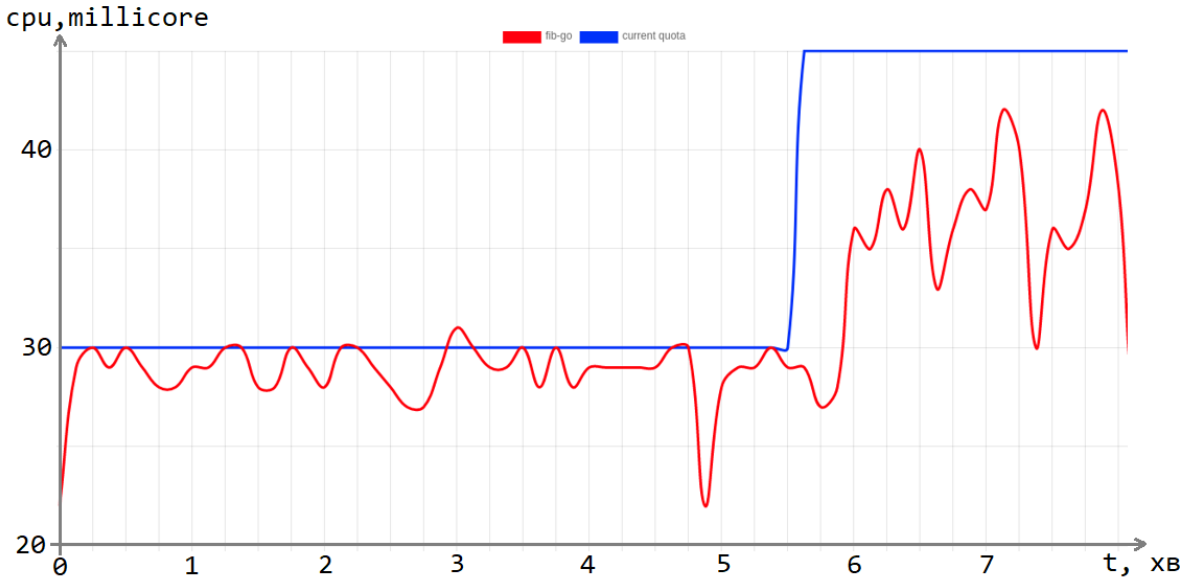

Рисунок 5.6 – Оптимізація використання процесора

5.3 Висновки

В даному розділі проведено тестування розробленої системи моніторингу керування обчислювальними ресурсами в кластері Kubernetes, а саме наступного функціоналу:

- моніторинг всіх існуючих мікросервісів у кластері, що розглянуто у розділі 5.1, а саме автоматичне знаходження розгортань у вказаних просторах імен, а також візуалізацію зібраних метрик по знайденим розгортанням.
- зменшення квот на ресурси при збитковому резервуванні, що протестовано на прикладі пам’яті та процесора у розділі 5.2;
- збільшення квот на ресурси при їх недостачі, що продемонстровано на прикладі використання процесора у розділі 5.2.

6 СТАРТАП ПРОЕКТ

За останнє десятиліття така форма підприємства як стартап набула попиту серед сучасних підприємців за рахунок спрощених вимог виходу на ринок. За рахунок розвитку нових каналів комунікації та неперервної інтеграції Інтернету у повсякденне життя як джерела інформації та реалізації послуг, також спрощуються процеси пошуку ресурсів та інвесторів на етапах становлення та розвитку стартап проекту. Серед основних факторів, які сприяють кількості росту інноваційних проектів на сучасному ринку, можна виділити інструменти пошуку цільової аудиторії, які набули значно більшої гнучкості, ніж на первинних етапах.

Проте, не дивлячись на широкий спектр лояльних умов, лише невелика частка стартап проектів дійсно зазнає успіхів. Така тенденція існує тому, що окрім завершеної ідеї проекту, дуже важливо сформувати реальну бізнес-модель, етапами якої є формування загальної концепції проекту, його виняткові риси, які створять попит, а також визначення цільової аудиторії.

Розроблення стартап проекту передбачає виконання набору кроків, які характеризують перспективи проекту на ринку, аналіз строків та етапи впровадження ідеї, фінансовий план і передбачення ризиків, а також кроки презентації концепції проекту інвесторам. Основні етапи і підходи створення стартап проекту будуть визначені у цьому розділі.

6.1 Опис ідеї проекту

В ході розробки стартап проекту була поставлена мета – автоматизувати процес управління і моніторингу споживчих ресурсів сервісів в рамках кластеру Kubernetes, що забезпечує мінімізацію мануальних налаштувань, мінімізує потребу у виділенні людських ресурсів для нагляду за системою, що значно зменшує вартість утримання системи сервісів і підтримує роботу системи в автономному режимі. Зміст ідеї, напрями застосування, а також їх переваги наведено у таблиці 6.1.

Таблиця 6.1 – Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
	1. Збір та аналіз метрик утилізації ресурсів	Автоматизування процесу моніторингу сервісів
	2. Забезпечення роботи системи в автономному режимі	Підтримка стабільності системи без залучення людських ресурсів
	3. Відстеження події аварійних завершень через нестачу ресурсів в кластері	Зменшення часу на пошук умов падіння продуктивності
	4. Можливість обчислення оптимальних лімітів для обчислювальних ресурсів	Автоматизація фази аналізу та обчислення необхідних ресурсів для сервісів

Під час проведення техніко–економічного аналізу проведено порівняння ряду характеристик стартап проекту з характеристиками систем конкурентів та аналогів. За критерії порівняння взято техніко–економічні характеристики ідеї, а саме визначення їх переліку для даного стартап проекту та для проектів–конкурентів.

Аналіз ринку показує, що у даної системи немає потенційних конкурентів. Аналогом може слугувати поєднання схожих рішень, наприклад, VPA та Goldilocks. Проте впровадження даного варіанту є досить складним, документація відсутня, а поріг для розуміння зазначених рішень є досить високим. Більш докладно описані слабкі, нейтральні та сильні сторони проектів у таблиці 6.2.

Таблиця 6.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

Техніко–економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
	Дана система	VPA	Magalix			
Ліцензія	Безкош-товна	Безкош-товна	Платна			+
Мова	Go	Go	Go		+	
Графічний інтерфейс	Є	Відсутній	Є			+
Робота в автомати-зованому режимі	Є	Є	Відсутня			+
Моніторинг подій	Є	Частково	Відсутня			+
Підтримка окремих подів	Відсутня	Є	Відсутня	+		

Продовження таблиці 6.2

Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
	Дана система	Vertical Pod Autoscaler	Magalix			
Патерни наванта-ження	Є	Відсутні	Відсутні			+

На основі зведеної таблиці порівняння техніко-економічних характеристик можна визначити слабкі, нейтральні та сильні сторони сформованої ідеї проекту стартапу. Аналізуючи співвідношення сильних сторін даного потенційного продукту з властивостями систем-конкурентів, можна зробити висновок про конкурентоспроможність даного продукту

6.2 Технологічний аудит ідеї проекту

Наступним етапом розробки стартап проекту є аудит технологій для визначення реалізації складових компонентів. Серед використаних технологій – мова програмування Go, для розгортання в кластері Kubernetes використовується фреймворк Operator SDK, для роботи з кластером використовується бібліотека від розробників Kubernetes – clientgo.

Дана технологічна сфера дуже швидко розвивається – одні проекти помирають і на зміну приходять нові. Саме тому дуже важливо обрати актуальні технології, для того, щоб проект не став застрілим з початку розробки. Обрані технології, а саме мова програмування Go і фреймворк Operator є новітніми та актуальними засобами.

В таблиці 6.3 проаналізовано всі використані при розробці компоненти для описання технологічної втілюваності ідеї:

Таблиця 6.3 – Технологічна здійсненність ідеї проекту

№	Ідея проекту	Технології її реалізації	Наявність технології	Доступність технологій
1		Мова програмування Go	Є в наявності	Доступні безкоштовно
2		Бібліотека для програмної взаємодії з кластером clientgo	Є в наявності	Доступні безкоштовно
3		Фреймворк для Розробки додатків Operator SDK	Є в наявності	Доступні безкоштовно
Обрані технології реалізації проекту включають в себе мову програмування Go, яка виконує взаємодію з кластером Kubernetes інструментами бібліотеки clientgo. Як командну утиліту для роботи з кластером обрано kubectl. Всі інструменти безкоштовні.				

6.3 Аналіз ринкових можливостей запуску

Дуже важливим підготовчим етапом перед впровадженням стартап проекту є аналіз ринкових можливостей та ринкових загроз. Виходячи зі стану розвитку ринку у даній галузі, можна сформувати напрямок розвитку, задовольняючи потреби потенційних клієнтів. Серед перешкод у реалізації проекту варто враховувати

потенційні ринкові загрози конкурентоспроможність даного проекту відносно набору пропозицій конкурентів(таблиця 6.4).

Можливості також можна знайти проаналізувавши замінні галузі. Наприклад, завдяки зменшенню авіаквитків, авіакомпанії можуть шукати можливості у споживчих сегментах, які в даний час постачаються іншими видами транспорту. Авіаперевізникам слід дослідити, скільки людей подорожує на міжміських автобусах та поїздах, які маршрути є найбільш затребуваними, скільки мандрівники платять за квитки, яка зайнятість міжміських автобусів та поїздів та що необхідно для переконати поточного пасажира автобусів чи поїздів вибрати замість цього подорож на літаку. Цей тип аналізу допомагає встановити конкурентні переваги порівняно з непрямими конкурентами та забезпечити розуміння додаткових можливостей для зростання.

Таблиця 6.4 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1.	Кількість головних гравців, од	12
2.	Загальний обсяг продаж, грн/ум.од	5 ум.од.
3.	Динаміка ринку (якісна оцінка)	Зростає
4.	Наявність обмежень для входу (вказати характер обмежень)	Початковий капітал до 30 тис. ум. од
5.	Специфічні вимоги до стандартизації та сертифікації	Формально відсутні

Продовження таблиці 6.4

№ п/п	Показники стану ринку (найменування)	Характеристика
6.	Середня норма рентабельності в галузі (або по ринку), %	78

Наступним кроком аналізу ринку є визначення актуальних потреб користувачів та відношення їх до груп цільової аудиторії. В залежності від потенційної групи цільових клієнтів, вимоги до споживчих товарів можуть змінюватись. Зведена таблиця характеристик потенційних клієнтів наведена в таблиці 6.5.

Таблиця 6.5 – Характеристика потенційних клієнтів стартап-проекту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Стрімкий розвиток розподілених систем	Компанії-постачальники програмного забезпечення	Доменна область поставки програмного забезпечення	Простота інтеграції до існуючої інфраструктури
2	Розподілення ресурсів сервісу від навантаження	Розробники програмного забезпечення на базі Kubernetes кластеру	Умови для застосування автоматизованої зміни лімітів для сервісу	Зручне відображення актуальних метрик

Визначивши потенційні групи клієнтів, проведено аналіз ринкового середовища. Таблиці 6.6 демонструє потенційні фактори загроз та розриває їх суть. У

таблиці 6.7 будуть наведені фактори можливостей, які сприяють майбутньому розвитку стартап проекту як конкурентоспроможної одиниці.

Таблиця 6.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1.	Поява більш продуктивних методів	Можлива поява та стрімкий розвиток нового методу, що матиме кращі характеристики	Спроба підвищити якість системи
2.	Відсутність інвестицій	Відсутність коштів унеможливлює розвиток за рахунок відсутності ресурсів	Пошук нових джерел коштів
3.	Ціна ліцензії	Об'єм ресурсів для розробки та тестування прямо пропорційно впливають на ціну товару	Перегляд ціни за рахунок пошуку більшої кількості покупців

Фактори можливостей - сприятливі фактори зовнішнього середовища, які можуть впливати на зростання бізнесу в майбутньому. Значення можливостей ринку для компанії в стратегічному плануванні: можливості ринку уособлюють джерела зростання бізнесу. Можливості необхідно аналізувати, оцінювати і розробляти план заходів по їх використанню з залученням сильних сторін компанії. Фактори які є актуальними для даного стартап проекту – це укладення угоди з розвиненими інформаційними компаніями, отримання інвестицій на розвиток проекту, а також

інтеграція з однією з розвинених компаній. В останньому випадку дана компанія буде зацікавлена в тому, щоб даний проект постійно розвивався. Інвестиції є

Таблиця 6.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1.	Підписання контракту	Укладення договору про інтеграцію даного стартап проекту в існуючу систему	Збільшення ресурсів для розробки
2.	Інвестиції	Отримання коштів від інвесторів	Вкладення інвестицій у нові технології
3.	Інтеграція з IT компаніє	Інтеграція в систему розвиненої компанії	Постійна фінансова підтримка

Як видно з проаналізованих даних, конкуренція як така відсутня, тобто є можливість вільно посісти це місце на ринку. Є також частково готові рішення: деякі необхідні бібліотеки є у вільному доступі, щоб використовувати їх для розробки конкретної ліцензованої програмної одиниці.

Основними перевагами при виборі цього товару є простота використання, широкий спектр функцій та якість роботи, якої немає у конкурентів.

Після аналізу конкуренції проводиться більш детальний аналіз умов конкуренції в галузі (за моделлю 5 сил М. Портера). Модель Michael Porter Five Force - простий, але ефективний інструмент аналізу бізнесу, який використовується для визначення того, чи може стратегія бути вигідною в конкурентному середовищі компанії. Правильно виконаний за допомогою відповідних інструментів, аналіз п’яти

сил може надати безцінне уявлення про ділову конкуренцію та про те, скільки сили можна отримати на ринку, щоб була можливість скорегувати свою стратегію успіху.

Кожна сила в моделі Майкла Портера представляє певний рівень товарної конкуренції: ринкова влада покупців, ринкова влада постачальників, загроза вторгнення нових учасників, ризик заміщення товарів, рівень конкуренції або внутрішньогалузева конкуренція.

Проведення ретельного аналізу конкурентів є важливим аспектом на початку будь-якого стартапу. Це дійсно важлива частина будь-якої успішної стратегії маркетингу. Дослідження того, що роблять ваші конкуренти, допоможе визначити їх сильні та слабкі сторони і надасть краще уявлення про те, чого слід уникати і що використовувати. Точне знання того, що відбувається у вашій галузі (наприклад, будь-які сучасні тенденції чи технології), також дозволить бізнесу бути більш динамічним та розвиватися відповідно до вимог споживачів. Аналіз п’яти сил Майкла Портера йде на крок далі і вимагає від замовника детального вивчення конкретних аспектів ринку, щоб замовник міг приймати більш стратегічні бізнес-рішення.

Далі у таблиці 6.8 проаналізовано риси впливу конкурентів на ринку на дії компанії.

Таблиця 6.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Вказати тип конкуренції – монополія/олігополія/ монополістична/чиста	олігополія	Орієнтованість на потреби клієнтів. Підтримання оновлень продуктів та вдосконалення функціональності

Продовження таблиці 6.8

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
За рівнем конкурентної боротьби - локальний/національний/…	міжнародний	Адаптація продукту під різні локалі
За галузевою ознакою - міжгалузева/ внутрішньогалузева	міжгалузева	Впровадження і бізнес різних галузей
За інтенсивністю	марочна	Маркетингова діяльність спрямована на заохочення клієнтів
Конкуренція за видами товарів: товарно-родова, товарно-видова, між бажаннями	Товарно-родова	Ведення конкуренції за рахунок задоволення потреб та маркетингової діяльності
За характером конкурентних переваг - цінова / нецінова	нецінова	Спрямування зусиль на створення якісної системи з гнучкою функціональністю

Привабливість розвитку на даному ринку додає досить слабка конкуренція та нешироκий спектр подібних продуктів. Також не виникає проблем з використанням частково підготовлених рішень, адже модулі запитів до кластеру надаються на основі безкоштовної ліцензії або умовно-безкоштовним модулем використання. Розвиток технологічності позиціонується як основна сильна сторона, на яку конкуренти не мають впливу.

Аналізуючи ступінь конкурентності на ринку, взято до уваги фактори маркетингового середовища, концепцію ідеї проекту, ключові потреби користувача та критерії їх вибору. Враховуючи цю інформацію, можна визначити конкурентні фактори. Фактори наведені у таблиці 6.9.

В таблиці 6.9 розглядаються фактори конкурентоспроможності, а саме наявність графічного інтерфейсу, легкість у використання, точність звітування та орієнтовність на потреби користувача. Орієнтовність на потреби користувача вважаю ключовою особливістю, адже конкуренти не можуть цього забезпечити. Легкість використання також є важливою рисою даного стартап-проекту.

Таблиця 6.9 – Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Зручність інтерфейсу	Проводячи аналіз конкурентних товарів, можна зробити висновки на основі відгуків користувачів та виділити недоліки, які не потрібно допускати та переваги, які можна прийняти до уваги на етапі розробки проекту.

Продовження таблиці 6.9

№	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
2	Забезпечення інтуїтивної зрозумілості	Так як система передбачає взаємодію з користувачем, буде забезпечено максимально лаконічний та зрозумілий інтерфейс з навігацією розділів.
3	Точність звітування	Для того щоб максимально задовольнити потребу адаптації системи до умов навантаження, точність звітування має бути високою
4	Орієнтованість на потреби користувача	Певні потреби користувачів досі залишаються не покритими, тому дана система виділяється серед представлених на ринку своє орієнтованістю на поточні потреби

Після визначення факторів конкурентоспроможності, виділено сильні та слабкі сторони стартап-проекту (СП – стартап-проект, К – конкурент). Дані наведені у таблиці 6.10.

Таблиця 6.10 – Порівняльний аналіз сильних та слабких сторін даного проекту

№ п/п	Критерій конкурентоспроможності	Оцінка 1-30	Порівняння існуючих рішень у конкурентів						
			-3	-2	-1	0	+1	+2	+3
1	Швидкість впровадження	30			+				
2	Якість результатів роботи проекту	25			+				
3	Додаткова конфігурація	27	+						
4	Звітність	25				+			

Наступним етапом аналізу ринку та можливості запуску проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities). SWOT-аналіз стартап-проекту наведено у таблиці 6.11.

Таблиця 6.11 – SWOT-аналіз стартап-проекту

Сильні сторони: - зручність інтерфейсу - економія - простота у використанні - врахування досвіду конкурентів	Слабкі сторони: - перший вихід на ринок - наявність open source рішень - ціна за рахунок складності розробки
Можливості: - інтеграція рішення у великі існуючі системи - використання стартапами	Загрози: - витіснення існуючими open source рішеннями - пошук клієнтів

6.4 Ринкова стратегія

Для побудови ринкової стратегії необхідно провести аналіз ринку: потреби клієнтів, потреба у даному продукті, конкурентів потенціалу ринку.

Під час аналізу ринку, насамперед, необхідно визначити товарно-географічні межі, суб’єкти ринку (продавці та клієнти), місткість ринку, а також структуру, особливо структуру суб’єктів господарювання на ринку, що характеризуються кількісними показниками, а також якісні показники структури ринку. Обов'язковим етапом проведення аналізу ринку є аналіз ринкового потенціалу ринкових суб'єктів.

Обираємо цільові групи, а саме IT компанії, стартап-проекти та державні IT організації. Всі вони зацікавлені в тому, аби зменшити економічні витрати. В перші два легкість входу найменша, а от в державному сегменті буде складніше.

Таблиця 6.12 – Цільові групи потенційних покупців

№ п/п	Цільові групи потенційних клієнтів	Зацікавленість введення проекту на ринок для споживача	Відсоток в межах цільової групи	Конкуренція в сегменті	Легкість входу у сегмент
1	IT компанії	Висока	70%	Середня, але згодом збільшиться	Низька складність
2	Стартап проекти	Середня	50%	Середня, але згодом збільшиться	Низька складність
3	Неприбуткові державні організації IT	Висока	20%	Низька	Середня складність

6.5 Висновки

Підбиваючи підсумки проведеного аналізу, можна дійти висновку, що стартап-проект на базі розробленого програмного рішення має потенційну можливість ринкового впровадження. Дана можливість підкріплюється великими шансами прибутковості, адже запроваджене рішення має достатньо переваг над представленими на ринку та може складати їм конкуренцію. Ринок автоматизованих рішень серед хмарних обчислень розвивається стрімкими темпами, адже попит на віртуалізацію сервісів все більше зростає. Користувачі Kubernetes шукають нові шляхи покращення та зростання продуктивності кластеру, а даний продукт покликаний саме для цього.

Були розглянуті такі показники, як сильні, слабкі та нейтральні характеристики проектів, технологічна здійсненність проекту, цільова аудиторія, попит ринку на рішення даної категорії, фінансові витрати, фактори загроз та фактори можливостей, шляхи отримання інвестицій та інше.

Подальший розвиток проекту є перспективним, враховуючи галузь впровадження, можливості, що пропонує даний проект, попит на ринку, а також відсутність монополії.

ВИСНОВКИ

В результаті виконання даної магістерської дисертації проведено детальний аналіз існуючих рішень, процесів розподілення ресурсів у кластері, алгоритму роботи планувальника Kubernetes, та на основі досліджень розроблено систему для автоматизації моніторингу та керування обчислювальними ресурсами у кластері Kubernetes.

В процесі огляду предметної області здійснено аналіз алгоритму роботи Kubernetes при розподіленні ресурсів пам’яті та процесора, на основі якого будуються алгоритми для обчислення оптимальних квот на ресурси. Оскільки Kubernetes застосовує різні підходи при розподіленні ресурсів пам’яті і процесора, то для кожного з них розроблено окремий алгоритм.

Алгоритм обчислення квоти на процесор дозволяє значно скоротити простоювання даного ресурсу за рахунок використання особливостей розподілення ресурсу та обмеження його використання в кластері Kubernetes.

Для оптимізації використання пам’яті розроблено декілька алгоритмів для різних типів навантаження. Один із алгоритмів дозволить додаткам з постійно зростаючим використанням пам’яті працювати без OOM помилок. Інші два націлені на економію даного ресурсу.

Основна частина цією роботи була присвячена саме автоматизації процесу керування – після розгортання розроблена система збирає метрики використання ресурсів для всіх видимих подів, проводить аналіз та надає оптимальні значення для всіх подів у цільовому просторі імен. При автоматичному режимі, система буде призначати обчислені значення самостійно, але обережно, щоб не порушити роботу додатку. Також реалізовано слідкування за подіями OOM, що свідчать про некоректно встановлені квоти.

Отже, розроблено систему, яка здатна значно спростити роботу адміністратору кластеру та розробникам. Крім того, дане рішення орієнтоване на економію ресурсів, що означає менші фінансові витрати.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ролик А.И. Управление корпоративной ИТ-инфраструктурой / А.И. Ролик, С.Ф. Теленик, М.В. Ясочка // К.: Наукова думка, 2018. – 576 с. https://acts.kpi.ua/app/uploads/2020/05/rolik_teleni_yasochka_uiti.pdf

2. Sam Newman. Building Microservices: Designing Fine-Grained System / Sam Newman – O'Reilly, 2015. – 251 с.

3. Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – O'Reilly, 2004. – 694 с.

4. Pethuru Raj Chelliah. Service Discovery and API Gateways - Essentials of Microservices Architecture, 2019

5. Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software / Eric Evans – Addison-Wesley, 2003. – 560 с.

6. Robert Martin. Clean Code: A Handbook of Agile Software Craftsmanship / Robert Martin – Addison-Wesley, 2008. – 465 с.

7. Mark Hamilton. Large-Scale Intelligent Microservices - [Електронний ресурс] – Режим доступу до ресурсу: www.researchgate.net/publication/344294726_Large-Scale_Intelligent_Microservices

8. Application Performance Isolation in Virtualization - [Електронний ресурс] – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/5284105

9. The Evolution of Container Usage at Netflix - [Електронний ресурс] – Режим доступу до ресурсу: https://netflixtechblog.com/the-evolution-of-container-usage-at-netflix-3abfc096781b

10. What Is Container Orchestration? - [Електронний ресурс] – Режим доступу до ресурсу: https://blog.newrelic.com/engineering/container-orchestration-explained/

11. Continuous integration vs. continuous delivery vs. continuous deployment - [Електронний ресурс] – Режим доступу до ресурсу: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

12. Andrew Tanenbaum. Modern Operating Systems / Modern Operating Systems – Pearson, 2015 – 1072 с.

13. Fundamentals of Virtualization, Containers, and Microservices ? - [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@pratyush.choudhury.bme16/fundamentals-of-enterprise-computing-virtualization-containers-and-microservices-4dfea6e645f2

14. Cgroups(7) - Linux manual page - [Електронний ресурс] – Режим доступу до ресурсу: https://man7.org/linux/man-pages/man7/cgroups.7.html

15. Large-scale cluster management at Google with Borg - [Електронний ресурс] – Режим доступу до ресурсу: https://research.google/pubs/pub43438/

16. Kubernetes. Manage HugePages - [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/tasks/manage-hugepages/scheduling-hugepages

17. Kubernetes best practices: Resource requests and limits - [Електронний ресурс] – Режим доступу до ресурсу: http://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits

18. Kubernetes. Managing Resources for Containers - [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

19. Kubernetes Resources Management. QoS, Quota, and LimitRange Ranges - [Електронний ресурс] – Режим доступу до ресурсу: https://www.cncf.io/blog/2020/06/10/kubernetes-resources-management-qos-quota-and-limitrangeb/

20. Kubernetes. Limit Ranges - [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/policy/limit-range/

21. Kubernetes Components - [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/components/

22. Kubernetes Metrics Options: Heapster vs. Prometheus vs. InfluxData - [Електронний ресурс] – Режим доступу до ресурсу: https://blog.containership.io/kubernetes-metrics-collection-options/

23. Capture and visualize metrics using Prometheus and Grafana - [Електронний ресурс] – Режим доступу до ресурсу: https://docs.particular.net/samples/logging/prometheus-grafana/

24. What is Prometheus? - [Електронний ресурс] – Режим доступу до ресурсу: https://prometheus.io/docs/introduction/overview/

25. The Kubernetes Scheduler - [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/cisco-emerging-technologies/the-kubernetes-scheduler-cd429abac02f

26. Kubernetes Scheduler 101 - [Електронний ресурс] – Режим доступу до ресурсу: https://www.magalix.com/blog/kubernetes-scheduler-101

27. OpenShift Scheduler Explained - [Електронний ресурс] – Режим доступу до ресурсу: https://docs.openshift.com/container-platform/4.5/nodes/pods/nodes-pods-vertical-autoscaler.html

28. Right-Sizing Pods with Vertical Pod Autoscaler - [Електронний ресурс] – Режим доступу до ресурсу: https://www.openshift.com/blog/how-full-is-my-cluster-part-4-right-sizing-pods-with-vertical-pod-autoscaler

29. KubeOptimizer - Plan and Automate Kubernetes Capacity Management - [Електронний ресурс] – Режим доступу до ресурсу: https://www.magalix.com/product/kubeoptimizer

30. Goldilocks: An Open Source Tool for Recommending Resource Requests - [Електронний ресурс] – Режим доступу до ресурсу: https://www.fairwinds.com/blog/introducing-goldilocks-a-tool-for-recommending-resource-requests

31. PYPL PopularitY of Programming Language - [Електронний ресурс] – Режим доступу до ресурсу: https://pypl.github.io/PYPL.html

32. Google Kubernetes Engine - [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/kubernetes-engine

33. Minikube Document - [Електронний ресурс] – Режим доступу до ресурсу: https://minikube.sigs.k8s.io/docs/start